

הסבר לקוד מטלה 3

Instructions for running the code:

- Maximum msg size should be at least 4.
- Project files are- server.py, client.py, packge.py, functions.py
- Get_packages.py- used for decoding packages that are exported from Wireshark as a JSON file.
- If there's a .txt file in the project directory it will use it automatically, if there are multiple text files you can choose which one to use (make sure to choose the same file on both sides of communication). If you want to get value from user input- remove all txt files from the project directory.
- When user enters the input, they automatically saved to a txt file in the project dir.

Package class:

```
import socket
import struct
import time
from typing import Dict, Optional

CLIENT_HEADERS = ["GET_MAX", "MSG", "CLOSE", "DONE"]
SERVER_HEADERS = ["ACK", "DISCONNECT", "RETURN_MAX", "TEMP"]
max_header_size = max( len(s) for s in CLIENT_HEADERS + SERVER_HEADERS)
PACKAGE_COUNT = 0
HEADER_SIZE = 34
```

Defining global variables-

CLIENT_HEADER, SERVER HEADER – for data validation when creating a new package. Separation between the client headers and the server header was made to be able to determine which of them created the package (important for package count recognition). Made globally for an easy change if needed, and making it accessible from anywhere in package.py.

max_header_size – for determining the maximal header length for fixed size encoding. Made globally since package class using it every time we encode or decode a package, by storing it globally we compute it once. Also keeps the code cleaner.

PACKAGE_COUNT – counting the number of packages a client created. By keeping it global, any new instance of Package can see and update the shared counter.

HEADER_SIZE – client and server can import the header size to keep it consistent in both side of communication. Also- prevents multiple changes in multiple files if we decide to change package header size. (note, header size refers to the byte length of all package fields beside payload).

```

15
16 @ class Package: 28 usages ▲ thelet +1*
17
18 @     def __init__(self, header: str, payload :str = "empty package"): ▲ thelet +1
19         global PACKAGE_COUNT
20         self.prev_seq = PACKAGE_COUNT
21         self.pos = PACKAGE_COUNT
22         if header in CLIENT_HEADERS :
23             self.seq = PACKAGE_COUNT
24             PACKAGE_COUNT += 1
25         elif header in SERVER_HEADERS :
26             self.seq = 0
27         else:
28             raise ValueError(f"Invalid header: {header}")
29         self.header = header
30         self.sent_time = time.time()
31         self.payload = str(payload)
32         self.ackrecv = False
33

```

Defining package fields-

Important note- fields prev_seq and seq are useless, and were used for debugging the code. Unfortunately, I didn't have enough time to remove them before submission deadline.

Pos- package sequence and the position of the payload in the original text. Used to recognize packages and to recreate the message in the right order.

Header- A string that identifies the type of package, helps to identify the purpose of each package and trigger the right function to handle the data. Also- when creating the package we use the header to identify the creator of the package and determine the sequence of the package. We also validate the header and return an error if it's invalid.

sent_time – timestamp for the package, updated before it sends (or resend), to recognize the time threshold.

payload – the actual data the package contains.

ack_recv = Boolean variable to indicate whether the client received an ack for this package.

```

38
39 @     # function to print the packet with its variables
40     def __str__(self):
41         PEP 8: E303 too many blank lines (5) : ...
42         return (
43             f"\n|  me: {self.sent_time} | ACK: {self.ackrecv} | "
44             f"prev seq: {self.prev_seq} | pos: {self.pos} | header: {self.header} |\n"

```

To string function-

For printing a package.

```

44 ######
45 # ACKs #
46 #####
47
48     def recvack(self): 1 usage ▲ thele
49         self.ackrecv = True
50
51
52     def send_ack(self, sock: socket, max_payload : int = 10): 2 usages ▲ thelet +1
53         Package.ackrecv = True
54         ack_package = Package(header: "ACK", str(self.get_pos()))
55         sock.send(ack_package.encode_package(max_payload))
56         print(f"ACK{str(self.seq)} sent!")
57

```

Ack functions-

recvack- called when a package was acked by the server, change the ack field to true.

send_ack – handles the sending of acks by the server. Gets the client socket to send the ack to, creates an ack package and sends it to the client.

```

def encode_package(self, max_payload) -> bytes: 9 usages ▲ thelet *
    global max_header_size
    """
    Format:
    - header: 10 bytes (string, padded with \x00)
    - payload: 50 bytes (string, padded with \x00)
    - seq: 4 bytes (integer)
    - sent_time: 8 bytes (double)
    - ack_state: 1 byte (boolean)
    - prev_seq: 4 bytes (integer)
    - pos : 4 bytes (integer)
    """
    # Pad or truncate the header/payload
    fixed_header = self.header[:max_header_size].ljust(max_header_size, fillchar: "\x00")
    fixed_payload = self.payload[:max_payload].ljust(max_payload, fillchar: "\x00")

    format_str = f"{max_header_size}s{max_payload}s i d ? i i"

    # Pack the data
    encoded = struct.pack(
        format_str,
        *fixed_header.encode("utf-8"),
        fixed_payload.encode("utf-8"),
        self.seq,
        self.sent_time,
        self.ackrecv,
        self.prev_seq,
        self.pos
    )
    return encoded

```

Encode function-

Encoding the package while keeping a fixed sized length. Using a defined format for padding the data by using python module "struct" and the method "pack".

Has a corresponding method for decoding . gets the maximal payload size from the user and returns a fixed sized encoded bytes data.

```
94     def decode_package(self, package_bytes: bytes, max_payload):  5 usages  ▲ thelet *
95         """
96             Deserialize from the fixed-size byte structure back into the Package fields.
97         """
98         global max_header_size
99         # Must match the same format used in encode_package
100        format_str = f"{max_header_size}s{max_payload}s i d ? i i"
101        expected_size = struct.calcsize(format_str)
102        if len(package_bytes) < expected_size:
103            raise ValueError(f"Expected {expected_size} bytes, but received {len(package_bytes)} bytes, max payload : {max_payload}")
104
105        #try:
106        unpacked_data = struct.unpack(format_str, package_bytes)
107        #except struct.error as e:
108        #    raise ValueError(f"Error unpacking data: {e} + max payload : {max_payload}")
109
110        # Unpacked data returns a tuple in the same order
111        raw_header, raw_payload, seq, sent_time, ack_state, prev_seq, pos = unpacked_data
112
113        # Decode strings and strip any trailing nulls
114        self.header = raw_header.decode("utf-8").rstrip("\x00")
115        self.payload = raw_payload.decode("utf-8").rstrip("\x00")
116        self.seq = seq
117        self.sent_time = sent_time
118        self.ackrecv = ack_state
119        self.prev_seq = prev_seq
120        self.pos = pos
121
```

Decode function-

Handles the decoding of the package and rebuild it from byte data. Like the encode method, using the python module "struct" and the method "unpack" to decode the fixed sized encoded package, according to the format. Gets the maximal payload size from the caller.

```
#####
# resend updates  #
#####

def update_for_resend(self, prev_seq, prev_pos):  1 usage  ▲ thelet
    global PACKAGE_COUNT
    self.sent_time = time.time()
    self.prev_seq = prev_seq
    self.seq = PACKAGE_COUNT
    self.pos = prev_pos
    PACKAGE_COUNT += 1

def get_package_for_resend(self, prev_seq : int, prev_pos : int):  ▲ thelet
    new_pack = Package(self.header, self.payload)
    new_pack.update_for_resend(prev_seq, prev_pos)
    return new_pack

def update_time(self):  1 usage  ▲ thelet
    self.sent_time = time.time()
```

Resend updates methods-

Used for updating the package before resending/sending. Unfortunately, I didn't have enough time to remove unnecessary parts. The only update that we used in the final code where time updates.

```
146 #####  
147 #      getters      #  
148 #####  
149  
150     def getSeq(self): 5 usages ▾ thelet  
151         return int(self.seq)  
152     def get_time(self): 2 usages ▾ thelet  
153         return self.sent_time  
154     def get_ack_state(self): 3 usages ▾ thelet  
155         return self.ackrecv  
156     def get_payload(self): 4 usages ▾ thele  
157         return self.payload  
158     def get_header(self): 7 usages ▾ thele  
159         return self.header  
160     def get_prev_seq(self): ▾ thelet  
161         return self.prev_seq  
162     def get_pos(self): 13 usages (2 dynamic) ▾ thelet  
163         return self.pos  
164
```

Getters functions.

Client Code:

Global variables-

```
19     HOST = '127.0.0.1'
20     PORT = 55558
21
22     HEADER_SIZE = package.HEADER_SIZE
23     MAX_MSG_SIZE = 4
24     BUFSIZ = HEADER_SIZE + MAX_MSG_SIZE
25     GOT_MAX_SIZE = False
26
27     ADDR = (HOST, PORT)
28     PARAMS : Dict[str,str] = {}
29     PACKAGES_TO_LOSE = [4,9,10]
30
31     CURRENT_PACKAGES : Dict[int,Package]= {}
32     LAST_ACK_SEQ : int = 0
33
34     TIME_WINDOW = 0
35     SEQ_WINDOW = 0
36
```

HOST, PORT- host is the IP used for communication, post is the server port.

HEADER_SIZE = explained in package class part.

MAX_MSG_SIZE – the maximal size of the first message the client sends to the server, when asking for the maximal message size.

PARAMS – dictionary that holds all the values the client gets from the file\user + the max size of message the server defined.

PACKAGES_TO_LOSE – list of the packages the client will lose to mimic the effect of lost packages.

CURRENT_PACKAGES – a dictionary with a key (package pos) and value (package) that saves all the sent packages (if a package was in packages to lose list, we still add to the Dict to mimic the losing of it).

LAST_ACK_SEQ – position of last ack the client received.

TIME_WINDOW, SEQ_WINDOW- reflects on window size and timeout, and changed while running according to the acks we receive. Time window holds the next time value where package threshold has passed. Seq window holds the end of the seq window we can send.

Starting the communication -

```
def create_client_socket(): 1 usage ▲ thele
    client_socket = socket.socket(AF_INET, SOCK_STREAM)
    client_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, value: 1)
    client_socket.connect(ADDR)
    initial_connection(client_socket)
    return client_socket

def initial_connection(client_socket): 1 usage ▲ thele *
    client_socket.send(Package(header: "GET_MAX", payload: "").encode_package())
    data = client_socket.recv(HEADER_SIZE + 4)
    pack_get = Package(header: "TEMP", payload: "")
    pack_get.decode_package(data, max_payload: 4)
    header = pack_get.get_header()
    if header == "RETURN_MAX":
        done = GET_MAX_Header(params_package=pack_get)
    return done
```

Creating a socket and initialize the connection with the server. Handle the request for max size before starting to receive and send data.

Receive function -

```
61 def receive(client_socket): 1 usage ▲ thele+1 *
62     """Continuously listens for messages (or ACKs) from the server."""
63     global PARAMS
64     global BUFSIZ
65     while True:
66         try:
67             print(f"current buffer : {BUFSIZ}")
68
69             data = client_socket.recv(BUFSIZ)
70             new_package = Package(header: "TEMP", payload: " ")
71             new_package.decode_package(data, MAX_MSG_SIZE)
72
73             header = new_package.get_header()
74
75             if header == "RETURN_MAX":
76                 GET_MAX_Header(params_package= new_package)
77
78             elif header == "ACK":
79                 ACK_Header(ack_package= new_package)
80
81             elif header == "DISCONNECT":
82                 print(f"Received DISCONNECT msg from server: {new_package} \n")
83                 CLOSE_Header(client_socket= client_socket)
84
85             elif not data:
86                 CLOSE_Header(client_socket= client_socket)
87                 break
88
89             else:
90                 print(f"Undetected header: \nRaw data received : {data}", flush=True)
```

```

92     except OSError as e:
93         # Check if the error is specifically WinError 10054
94         if hasattr(e, 'winerror') and e.winerror == 10054:
95             print(f"Server forcibly closed the connection", flush=True)
96         else:
97             print(f"Error while receiving data: {e}", flush=True)
98             CLOSE_Header(client_socket=client_socket)
99             break

```

Note- this function is called from a different thread, and handles the received data while we send packages to the server.

Function gets the package and decode it, than recognizes it's type and calls the function to handle it.

Handling packages – by headers

Header "RETURN_MAX"

```

122     def GET_MAX_Header(params_package : Package): 2 usages ▲ thelet +1*
123         global PARAMS
124         global TIME_WINDOW
125         global SEQ_WINDOW
126         global BUFSIZ
127         global HEADER_SIZE
128         global GOT_MAX_SIZE
129         PARAMS.update(functions.get_client_params())
130         PARAMS.update({"maximum_msg_size" : params_package.get_payload()})
131         print(f" got max size from server: {PARAMS}")
132         TIME_WINDOW = float(time.time()) + float(PARAMS["timeout"])
133         SEQ_WINDOW = int(PARAMS["window_size"])
134         update_buffer_andmax_size(int(PARAMS["maximum_msg_size"]))
135         GOT_MAX_SIZE = True
136         return True
137

```

Function to handle the server response for the **max msg size request**, Calls when server response to the request, with a package that it's header if "RETURN_MAX".

After receiving the response, we update the params dictionary , and the time and seq window.

Header "ACK"

```
136     def ACK_Header(ack_package : Package): 1 usage ▲ thelet +1 *
137         global LAST_ACK_SEQ
138         global TIME_WINDOW
139         global SEQ_WINDOW
140         acked_pack = CURRENT_PACKAGES.get(int(ack_package.payload))
141         if acked_pack is not None:
142             LAST_ACK_SEQ = get_last_ack_seq()
143             if acked_pack.get_pos() > LAST_ACK_SEQ +1:
144                 print(f"ACKED PACKAGE OUT OF ORDER: {acked_pack}, acks {[i for i in range(LAST_ACK_SEQ+1, acked_pack.get_pos())]}")
145                 for i in range(LAST_ACK_SEQ+1, acked_pack.get_pos()):
146                     next_ack = CURRENT_PACKAGES.get(i)
147                     print(f"Correcting lost ack {next_ack.get_pos()}")
148                     next_ack.recvack()
149
150             acked_pack.recvack()
151
152             print(f"\nreceived ACK {ack_package.payload} ! \n")
153             update_window_size()
154         else:
155             print(f"Warning: No package found for key '{ack_package.payload}' .")
```

A function that handles the receive of **acks**, with the header "**ACK**". updates the time window and the seq window (by calling the function "update_window_size", finds the package that received an ack in the CURRENT_PACKAGES dictionary and mark the filed recv_ack as true. If client receives an ack out of order (bigger than last acked pack+1) - it will assume all the packages between this ack and the previous ack it received where received in the server and mark them as acked. Since the server sends acks only if all the previous packages arrived, this will handle the case of a lost ack correctly.

Header "DISCONNECT"

```
154     def CLOSE_Header(client_socket : socket.socket): 3 usages ▲ thele +1
155         print(fr"Closing connection...")
156         sleep(1)
157         print("\nall packages sent: ")
158         for seq in CURRENT_PACKAGES:
159             print(CURRENT_PACKAGES.get(seq))
160
161         client_socket.close()
162         sys.exit(0)
163
```

Handles closing of client socket when a "**DISCONNECT**" header is received. The server sends a DISCONETCT package as a response to a "CLOSE" package from the client, to approve the client that it got the request to close the connection and finish the communication.

Sending of data logic-

```
250     def send_from_text_file(client_socket : socket.socket): 1 usage ▲ thelet
251         msg = PARAMS.get("message")
252         print(f"sending msg from file: {msg}")
253         sliced_msg = slice_data(msg.encode("utf-8"))
254         print(f"sliced msg: {sliced_msg}")
255         send_logic(client_socket, sliced_msg)
256
257         send_CLOSE_msg(client_socket)
258         time.sleep(1)
259
260
261     def send_logic(client_socket : socket.socket, sliced_msg : list[bytes]): 1 usage ▲ thelet
262         seq = 0
263         for data_slice in sliced_msg:
264             seq+=1
265             update_window_size()
266             time.sleep(0.3)
267             if not check_time_threshold() or not check_seq_threshold(seq):
268                 print("resend lost package")
269                 sent = resend_logic(client_socket)
270                 send_data(data_slice.decode("utf-8"), client_socket)
271             before_closing(client_socket)
272
273
```

Those functions handle the main logic of sending the data.

Send_text_from_file – takes the msg parameter from PARAMS dict, using the slice function to slice the data according to the maximal message size, and transfer the sliced data to send_logic() function.

Send_logic- gets the list of data and handles it's transfer to the send_data or resend_data. Check the threshold situation before sending data, in order to stop the sending if a threshold has passed. If the threshold have passed we first handle the lost package and then sends the data. If not- we send the data.

After sending all the data we call for a function that handle the remaining lost packages before closing the socket.

```
199     def send_data(data_slice : str, client_socket): 1 usage ▲ thelet
200         global PACKAGES_TO_LOSE
201         print(f"Time window: {TIME_WINDOW} Seq window: {SEQ_WINDOW}")
202         new_pack = Package(header: "MSG", data_slice)
203         print(f"SENDING package :\n {str(new_pack)}")
204         time.sleep(0.5)
205         CURRENT_PACKAGES.update({int(new_pack.get_pos()): new_pack})
206         print(f"added package to current packages: {new_pack}")
207         f"current packages state: {[str(pack) for pack in CURRENT_PACKAGES.values()]}"
208
209         if int(new_pack.get_pos()) in PACKAGES_TO_LOSE:
210             print(f"package {new_pack.get_pos()} will be lost")
211             PACKAGES_TO_LOSE.remove(int(new_pack.get_pos()))
212         else:
213             client_socket.send(new_pack.encode_package(int(PARAMS["maximum_msg_size"])))
```

Function to send the packages to the server. Adds the package to the CURRENT_PACKAGES dict and sends the package. If the package pos is in PACKAGES_TO_LOSE, we skip the sending part to mimic a lost package (and removing the pos to allow the package to be resend when needed).

Resending packages-

```
220
229     def resend_logic(client_socket : socket.socket): 2 usages ▾ thelet
230         lost_pack = get_lost_package()
231         if lost_pack is not None:
232             while True:
233                 print(f"transfer for resending package: {lost_pack}")
234                 resend_data(lost_pack, client_socket)
235                 worked = wait_for_ack(lost_pack.get_pos())
236                 if worked:
237                     break
238             return True
239         else:
240             print(f"no lost package found, skipping resend")
241         return False
```

```
190
191     def resend_data(package : Package, client_socket): 1 usage ▾ thelet
192         print(f"Time wind: {TIME_WINDOW} Seq win: {SEQ_WINDOW}")
193         print(f"RESENDING package :\n {package}")
194         time.sleep(0.5)
195         CURRENT_PACKAGES.get(package.get_pos()).update_time()
196         client_socket.send(package.encode_package(int(PARAMS["maximum_msg_size"])))
```

resend_logic -called when we recognized that a threshold have passed. Gets the lost package by using the function get_lost_packages(), use the resend_data() function to resend the package and then calls the function wait_for_ack() to hold the sending until the server sends an ack for this package (and for the next packages). If wait_for_aks() returns true- we received an ack and can return to sending new packages. If it returned false- we repeat this process.

Waiting for ack-

```
260     def wait_for_ack(pos: int): 1 usage ▲ thelet
261         while True:
262             last_ack = get_last_ack_seq()
263             seconds = float(PARAMS["timeout"])
264             print(f"waiting for ack: {pos}")
265             while seconds > 0 and int(last_ack) < int(pos):
266                 print(f"time left: {seconds}", end="")
267                 time.sleep(0.3)
268                 seconds -= 0.3
269                 last_ack = get_last_ack_seq()
270                 if int(last_ack) >= int(pos):
271                     return True
272                 elif seconds <= 0:
273                     return False
274                 else:
275                     continue
```

Wait for ack- sets a timer that counts down from time threshold. Checks if we received an ack by using the function get_last_ack_seq(). if get_last_ack_seq() returns the pos of the package we are waiting for (or a bigger pos), function returns true. If the timer is over and we didn't receive an ack, function returns false.

Disconnecting from the server-

Before disconnecting from the server-

```
278     def before_closing(client_socket : socket.socket): 2 usages ▲ thele+1
279         print("handling lost packages before close:")
280         while not all_acks_received():
281             time.sleep(0.5)
282             while get_lost_package():
283                 resend_logic(client_socket)
285         else:
286             break
287
```

Before sending the "CLOSE" package, we want to make sure all the packages have been acked. We wait for 0.5 seconds to see if an ack will arrive. If not- while the function get_lost_package() returns a value (which means there's still unacked packages) we call resend_logic().

Disconnecting-

```
161     def send_CLOSE_msg(client_socket : socket.socket): 1 usage ▲ thelet +1
162         before_closing(client_socket)
163         print("finish current transfer:")
164         finish_package = Package( header: "DONE", payload: "EOMsg")
165         CURRENT_PACKAGES.update({finish_package.getSeq(): finish_package})
166         client_socket.send(finish_package.encode_package(int(PARAMS["maximum_msg_size"])))
167         while True:
168             lost_pack = get_lost_package()
169             seconds = float(PARAMS["timeout"])
170             while seconds > 0 and lost_pack is not None:
171                 print(f"Time left: {seconds} seconds")
172                 time.sleep(0.2)
173                 seconds -= 0.2
174                 lost_pack = get_lost_package()
175             if lost_pack is not None:
176                 print(f"lost pack: {get_lost_package().getSeq()}")
177                 print("resending DONE msg")
178                 client_socket.send(finish_package.encode_package(int(PARAMS["maximum_msg_size"])))
179             else:
180                 break
181
182         close_package = Package( header: "CLOSE", payload: "request to close connection")
183         CURRENT_PACKAGES.update({close_package.getSeq(): close_package})
184         client_socket.send(close_package.encode_package(int(PARAMS["maximum_msg_size"])))
185
```

Function sends a finish package "DONE" that indicates all data was sent. Then enters a loop that waits for ack on this package. Each loop we wait "timeout" seconds, and if we didn't receive an ack, we try to resend. After receiving an ack, we send "CLOSE" package to the server.

Note that the actual closing of the socket happens after receiving a "DISCONNECT" package from the server, that's approves he received client's request to disconnect.

More functions-

Functions to check if threshold passed-

```
100
101     def check_time_threshold(): 1 usage ▲ thelet +1
102         if package is not None:
103             if TIME_WINDOW and TIME_WINDOW <= time.time():
104                 print(f"THRESHOLD PASSED (time): \nTime Window: {TIME_WINDOW}, Current time: {time.time()}")
105                 return False
106             else:
107                 return True
108
109     def check_seq_threshold(package_seq): 1 usage ▲ thelet +1
110         if package is not None:
111             if SEQ_WINDOW and int(SEQ_WINDOW) < int(package_seq):
112                 print(f"THRESHOLD PASSED (seq): \nSeq Window: {SEQ_WINDOW}, Current pack seq: {package_seq}")
113                 return False
114             else:
115                 return True
```

Check_time_threshold – returns false if current time is bigger than the current window(means that we passed the time window for sending packages). Else, returns true.

Check_seq_threshold – gets the next seq of the package we want to send and returns false if it's bigger than the current window (means that we passed the seq window for sending packages). Else, returns true.

```

306     def update_time_window(): 1 usage  ▲ thelet
307         global TIME_WINDOW
308         last_no_ack = get_lost_package()
309         if last_no_ack is not None and TIME_WINDOW is not None:
310             print(f'Updating TIME window by package {last_no_ack.get_pos()} :\n'
311                  f'New time window {float(last_no_ack.get_time()) + float(PARAMS["timeout"])} , Prev time window: {TIME_WINDOW}')
312             TIME_WINDOW = float(last_no_ack.get_time()) + float(PARAMS["timeout"])
313         else:
314             print("starting time window based on current time ")
315             TIME_WINDOW = float(time.time()) + float(PARAMS["timeout"])
316
317
318     def update_seq_window(): 1 usage  ▲ thelet *
319         global SEQ_WINDOW
320         last_ack = get_last_ack_seq()
321         if last_ack is not None and SEQ_WINDOW is not None:
322             print(f'Updating SEQ window size by last acked pack: {last_ack}\n'
323                  f'New seq window: {int(PARAMS["window_size"]) + int(last_ack)} Prev seq window : {SEQ_WINDOW}\n')
324             SEQ_WINDOW = int(PARAMS["window_size"]) + int(last_ack)
325         else:
326             print("starting seq window based on params ")
327             SEQ_WINDOW = int(PARAMS["window_size"])

```

Functions to update the thresholds-

Update_time_window – function to update the time window base on the oldest package that didn't receive an ack. Find this package by using get_lost_package() function (returns the oldest package that didn't receive an ack). Takes the timestamp of the package (send time) and adds our "timeout" parameters to it, the result is the window of time we can send packages until receiving an ack for this pack. Whenever an ack is received, we call this function again to update the current window. If it's the first package we send (last_no_ack returns None), we update the window to "current time" + timeout (happens when the function is called for the first time, when sending the first package).

Update_seq_window – same logic applies here. We update by the last ack we received (if we didn't receive an ack yet, we update based on package pos '0'). We calculate the seq window by "last ack sequence" + "window_size".

```

300
301     def update_window_size(): 2 usages  ▲ thelet
302         print(f"\nupdated window size: ")
303         update_time_window()
304         update_seq_window()

```

A small function to call both of those functions together when updating a window.

Function to find the oldest package that didn't receive an ack-

```
207
290     def get_lost_package(): 6 usages ▲ thelet
291         global CURRENT_PACKAGES
292         print(f"CURRENT PACKAGES: {[str(CURRENT_PACKAGES.get(key)) for key in CURRENT_PACKAGES]}")
293         # Find the smallest key where the package has not received an ACK
294         for key in sorted(CURRENT_PACKAGES, reverse=False):
295             pack = CURRENT_PACKAGES.get(key)
296             if not pack.get_ack_state():
297                 print(f"min pos, no ack: {str(pack)}\n")
298                 return pack
299         return None
```

Function sorts CURRENT_PACKAGES from the smallest key to the largest.

Search for the first package that didn't receive an ack and returns it.

Note that it's not the most efficient way to do that, but it's a reliable. And since we are handling small data, and considering we are using multiple threads which can cause different kind of issue, and since we preferred that our code will be accurate – we ended up using this system.

Function to find the last package that received an ack-

```
330     def get_last_ack_seq(): 4 usages ▲ thelet
331         for seq in sorted(CURRENT_PACKAGES, reverse=True):
332             if CURRENT_PACKAGES.get(seq).get_ack_state():
333                 return seq
334         return 0
```

Used a similar method to get_lost_package() function. Returns the package with the largest key that received an ack.

Function to slice the message according to maximum_msg_size-

```
238
239     def slice_data(data : bytes): 1 usage ▲ thelet
240         chunks = []
241         for i in range(0, len(data), int(PARAMS["maximum_msg_size"])):
242             chunks.append(data[i:i + int(PARAMS["maximum_msg_size"])])
243         return chunks
244
```

Function to check if we received acks for all the packages-

```
713
74     def all_acks_received(): 1 usage ▲ thelet +1
75         global CURRENT_PACKAGES
76         return all(pkg.get_ack_state() for pkg in CURRENT_PACKAGES.values())
77
```

Function to update our recv buffer after receiving maximum_msg_size-

```
337     def update_buffer_andmax_size(new_max_size): 1 usage ▲ thelet
338         global MAX_MSG_SIZE
339         global BUFSIZ
340         MAX_MSG_SIZE = new_max_size
341         BUFSIZ = HEADER_SIZE + MAX_MSG_SIZE
342         print(f"updated buffer size to {BUFSIZ} and max size to {MAX_MSG_SIZE}", flush=True)
343
```

Server Code:

Global variables-

```
22  
23     MAX_CLIENTS = 1  
24     ADDR = (HOST, PORT)  
25     CLIENTS = []  
26     PARAMS = functions.get_server_params()  
27     print(PARAMS)  
28     MAX_MSG_SIZE = 4  
29     LAST_SEQ=0  
30     HEADER_SIZE = package.HEADER_SIZE  
31     BUFSIZ = HEADER_SIZE + MAX_MSG_SIZE  
32     ACKS_TO_LOSE = [0]  
33
```

HOST, PORT, ADDR – same as in client.

MAX_CLIENTS- I used a base template I made for a project with a server that handles multiple clients, here we only need to handle one, but I didn't delete the variable in case it will be useful in the future.

CLIENTS- from the same template, list to store all clients addresses.

PARAMS – receives max size from a function that handles the receiving of the parameter.

MAX_MSG_SIZE – initiates to the "GET_MAX" payload length, since it's the first package the server receives from the client.

LAST_SEQ – the position (sequence number) of the last package socket send an ack on.

HEADER_SIZE, BUFSIZ – same as client

ACKS_TO_LOSE – list of package sequences the server won't send acks for, but will treat them as they where received and acked. This will mimic the effect of a lost acked package on the communication.

Starting the communication -

Creating the socket-

```
66 33     def create_server_socket(): 1 usage ▲ thele +1
67 34         try:
68 35             SERVER_SOCKET = socket.socket(AF_INET, SOCK_STREAM)
69 36             SERVER_SOCKET.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, value: 1)
70 ...
71 37             SERVER_SOCKET.bind(ADDR)
72 38             SERVER_SOCKET.listen(MAX_CLIENTS)
73 39             print("creating server....")
74 40             return SERVER_SOCKET
75
76 41
77 42         except Exception as e:
78 43             print(f"Error while creating the server: {e}")
```

Connecting to the client-

```
def accept_incoming_connections(server_socket : socket.socket): 1 usage ▲ thele +1
    """Sets up handling for incoming clients."""
    print("Waiting for connections....")
    global CLIENTS
    CLIENTS.clear()
    try:
        while True:
            try:
                client, client_address = server_socket.accept()
                if client_address not in CLIENTS:
                    CLIENTS.append(client_address)
                    print(f"{client_address} has connected.", flush=True)
                    # Handle the client in a separate thread

                    client_THREAD = Thread(target=handle_client, args=(client,client_address))
                    client_THREAD.start()
                else:
                    print("Already connected.", flush=True)
                    break
            except OSError as e:
                if hasattr(e, 'winerror') and e.winerror == 10054:
                    print(f"Client: forcibly closed the connection", flush=True)
                else:
                    print(f"Error while connecting to client: {e}", flush=True)
                    break

    except Exception as e:
        print(f"Error while connecting to client: {e}", flush=True)
```

Starting by adding the client to clients list and handle duplicated client error- I used a template from another project I wrote for multiple clients, this part is not relevant to this exercise, but didn't remove it in case it will be useful in the future. After accepting the client we start a thread to handle the client. Note that the accept_incoming_connections method also called from a thread- in order to handle multiple clients. Again, leftovers from my template.

Receive logic –

```
def handle_client(CLIENT_SOCKET, client_address):  # usage -> thelet +1 *
    global MAX_MSG_SIZE
    print(f"Handling client {client_address}", flush=True)
    done= initial_connection(CLIENT_SOCKET, client_address)
    if done:
        update_buffer_andmax_size(PARAMS["maximum_msg_size"])
    try:
        while True:
            try:
                data = CLIENT_SOCKET.recv(BUFSIZ)
                if not data:
                    print(f"Client {client_address} forcibly closed the connection", flush=True)
                    CLOSE_Header(client_socket= CLIENT_SOCKET, client_address=client_address)
                    break

                new_package = Package(header= "TEMP", payload= " ")
                print(f"current max size: {MAX_MSG_SIZE}")
                new_package.decode_package(data, MAX_MSG_SIZE)
                header = new_package.get_header()

                if header == "GET_MAX":
                    print(f"GET_MAX from {client_address}")
                    GET_MAX_Header(client_socket= CLIENT_SOCKET)

                elif header == "MSG" or header == "DONE":
                    print(f"MSG from {client_address}, starting to receive:")
                    MSG_Header(client_socket= CLIENT_SOCKET, msg_package= new_package, client_address= client_address)

                elif header == "CLOSE":
                    print(f"CLOSE request from {client_address}")

11               else:
12                   print(f"Undetected header: \nRaw data received from {client_address}: {data}", flush=True)
13                   pass
14
15           except OSError as e:
16               # Check if the error is specifically WinError 10054- means that client disconnected without sending CLOSE pac
17               if hasattr(e, 'winerror') and e.winerror == 10054:
18                   print(f"Client {client_address} forcibly closed the connection", flush=True)
19                   break
20               else:
21                   print(f"Error with client {client_address}: {e}", flush=True)
22                   break
23           except OSError as ex:
24               print(f"Error while handling client {client_address}: {ex}", flush=True)
```

Starting by calling the function `initial_connection()`, that handles the transfer of the `maximum_msg_size` data. Returns true after transferring without error.

After `initial_connection` returns true, we update our buffer and `max_msg_size` to the correct data we received from file or from user.

Than, until we receive an error or a request to disconnect, we call `recv(buffsize)` function, decoding the received byte data to a package, and handle the package according to the header.

Initialize connection-

```
120     def initial_connection(client_socket : socket.socket, client_address): 1 usage ▲ thelet
121         data = client_socket.recv(HEADER_SIZE + 4)
122         pack_get = Package( header: "TEMP", payload: "")
123         pack_get.decode_package(data, max_payload: 4)
124         header = pack_get.get_header()
125         if header == "GET_MAX":
126             print(f"GET_MAX from {client_address}")
127             done = GET_MAX_Header(client_socket=client_socket)
128             return done
129
```

Handles the receiving of the first package- the request for maximum size of message. Fix payload size on both sides of communication is 4.

Handling packages – by headers

"GET_MAX" header-

```
128     def GET_MAX_Header(client_socket : socket.socket): 2 usages ▲ thelet +1
129         global MAX_MSG_SIZE
130         try:
131             new_package = Package( header: "RETURN_MAX", PARAMS["maximum_msg_size"])
132             print(f"SENDING max size package :\n {new_package}")
133             client_socket.send(new_package.encode_package(4))
134             return True
135         except Exception as e:
136             print(f"Error while sending max size package: {e}", flush=True)
137
```

"GET_MAX" header is a request from the client to get the max size of msg. server creates a new package where payload is the max size of msg. encode the package with fix size of 4 for the payload (same as client), and sends the encoded package to the client.

"CLOSE" header-

```
def CLOSE_Header(client_socket : socket.socket, client_address): 3 usages ▲ thelet +1
    print(f"Client {client_address} disconnected", flush=True)
    client_socket.send(Package( header: "DISCONNECT", payload: "approving client disconnection").encode_package(int(PARAMS["max_size"])))
    client_socket.close()
    CLIENTS.remove(client_address)
```

"CLOSE" header is a request from a client to disconnect. Server sends an approval, than closes the client socket on his side. (and removes client from clients list, added this part to match with adding the client to this list, even though it not relevant to this exercise).

"MSG" header-

This is the main logic of receiving the data and sending the acks.

Part 1- Variables

```
def MSG_Header(client_socket : socket.socket, msg_package : Package, client_address : tuple):
    last_seq = int(msg_package.get_pos()) - 1
    msg_list = []
    full_msg = []
    global ACKS_TO_LOSE
```

msg_list - a list that saves all the packages server received.

full_msg – A list containing all the packages that the server has acknowledged.

Last_seq – The last sequence number that the server acknowledged.

Global ACKS_TO_LOSE – a list of packages the server will not acknowledge, to mimic the effect of lost ack package.

+ client_socket and the first package are passed as arguments to the function.

Part 2-

```
47     while msg_package.get_header() == "MSG":
48         try:
49             print(f"received package: {msg_package} last_seq = {last_seq}")
50             msg_list.append(msg_package)
51             msg_list.sort(key=lambda package: package.get_pos())
52
53             for pack in msg_list:
54                 if int(pack.get_pos()) == last_seq + 1:
55                     print(f"sending ack for pack: {pack}")
56                     if int(pack.get_pos()) in ACKS_TO_LOSE:
57                         ACKS_TO_LOSE.remove(int(pack.get_pos()))
58                     else:
59                         pack.send_ack(client_socket, PARAMS["maximum_msg_size"])
60                         last_seq = pack.get_pos()
61                         full_msg.append(pack)
```

We start by adding the current package received by the server to the **msg_list**.

This ensures that, in case a package is lost, the packages received prior to the resending of the lost package are stored and can be processed once the lost package is successfully resent and received.

Then we sort msg_list by key to make sure we will send the right acks in the right order if needed (note that this list should be sorted or almost sorted at all time, before we resort it).

Then, the server searches the **msg_list** for the next sequence that needs to be acknowledged. If found, in case this pos is not in ACKS_TO_LOSE list -the server sends an ack for this package- else, the server will skip sending the ack. Then-updates the **last_seq**, and adds the package to the **full_msg** list. Server will

continue to check msg_list in case there are more packages that needs to receive an ack. This approach ensures that acknowledgments are sent in the correct order, as specified by the exercise instructions.

We remove the acknowledged packages from msg_list after adding them to full_msg.

Part 3- receiving the next package

```
    data = client_socket.recv(BUFSIZ)
    msg_package = Package( header: "TEMP", payload: " " )
    msg_package.decode_package(data, PARAMS["maximum_msg_size"])
except Exception as e:
```

Part 4- handling end of message

```
167
168     if msg_package.get_header() == "DONE":
169         print(f"done sending file")
170         if msg_list or len(msg_list) > 0:
171             print(f"warning! lost data: {[pack.get_payload() for pack in msg_list]}")
172         MSG_DONE_Header(client_socket, msg_package, full_msg)
173
```

When a "DONE" header received, it means the client finished sending data, and we call to the function that handles "DONE" headers.

"DONE" header-

```
92     def MSG_DONE_Header(client_socket : socket.socket, msg_package : Package, msg : List[Package]): 1
93         if msg_package:
94             msg_package.send_ack(client_socket, PARAMS["maximum_msg_size"])
95             str_msg = ""
96         if msg:
97             msg.sort(key=lambda package: package.get_pos())
98             print(f"msg packages: {[pack.get_payload() for pack in msg]}")
99             for pack in msg:
100                 str_msg += pack.get_payload()
101             print("\n full message received: ")
102             print(str_msg)
103             print("")
```

Sends ack for the "DONE" package. Than, recreates the message string for the sorted packages list and print it.

Getting values from .txt\user:

Function to chose how to get values-

```
8  def get_params() -> Dict[str, Any]:  1 usage  ▲ thelet
9      try:
10          paths = find_all_text_files()
11          if len(paths) == 1:
12              return file_to_json(paths[0])
13          if len(paths) >1:
14              path = choose_text_file(paths)
15              return file_to_json(path)
16          if len(paths) == 0:
17              print("No text files found")
18              params = get_from_user()
19              write_dict_to_file(params, filename: "params.txt")
20              print("params saved to params.txt")
21          return file_to_json("params.txt")
22      except Exception as error:
23          print(error)
24
```

1. If there's only 1 txt file in project path- gets values from this file.
2. If there's more than 1 txt file- allow user to choose which file to use- allows to create multiple test cases ahead, and choose which one of them you want to run.
3. If there's no txt file in project path- gets value from user.

Get from user function-

```
28
29  def get_from_user():  1 usage  ▲ thele +1
30      message = input("Enter a message: ")
31      max_msg_size = input("Enter the maximum message size (in bytes): ")
32      window_size = input("Enter the window size: ")
33      timeout = input("Enter timeout value (in seconds): ")
34      params = {
35          "message": message,
36          "maximum_msg_size" : max_msg_size,
37          "window_size" : window_size,
38          "timeout" : timeout
39      }
40      try:
41          validate_input(params)
42          return params
43      except Exception as e:
44          print(f"Unvalied parameters from user: {e}")
45          sys.exit(1)
```

Function that receives the values from user input. The main function-`get_params()` calls this function if no txt files found, and saves the user input to a text file called "params".

Get from file function-

```
45
46     def file_to_json(file_path: str) -> dict: 3 usages ▲ thelet
47         try:
48             json_data = {}
49
50             with open(file_path, 'r') as file:
51                 for line in file:
52                     # Skip empty lines or lines without a colon
53                     if not line.strip() or ':' not in line:
54                         continue
55
56                     # Split the line into key and value
57                     key, value = line.strip().split(sep=":", maxsplit=1)
58                     key = key.strip()
59                     value = value.strip()
60
61                     # Convert numeric values to integers
62                     if value.isdigit():
63                         value = int(value)
64
65                     json_data[key] = value
66
67             return json_data
68         except Exception as e:
69             raise ValueError(f"Error processing file: {e}")
70
```

takes a file path as input and returns the file's content as a dictionary (JSON-like format). Reads the txt file line by line and uses ":" as the delimiter.

Validating user input-

```
72     def validate_input(params : Dict[str, Any]) -> None: 1 usage ▲ thele +1
73         try:
74             for key in params:
75                 if key == "message":
76                     pass
77                 elif not params[key].isnumeric():
78                     raise ValueError("all values must be numeric")
79                 elif int(params[key]) <= 0:
80                     raise ValueError("all values must be positive")
81         except Exception as e:
82             print(f"Error while validating parameters: {e}")
83
```

Function that validates that the user input is in the current format.

Finding txt file-

```
85
86     def find_all_text_files() -> list[str]: 1 usage  ± thele +1*
87         try:
88             current_directory = os.getcwd()
89             text_files = []
90
91             # Iterate over all files in the current directory
92             for file_name in os.listdir(current_directory):
93                 file_path = os.path.join(file_name)
94
95                 # Check if it's a .txt file
96                 if os.path.isfile(file_path) and file_name.endswith('.txt'):
97                     text_files.append(file_path)
98
99             return text_files
100        except Exception as e:
101            print(f"Error while searching for text files: {e}")
102
```

Function that returns all the txt files in the project directory. If you want to use a file from different directory- replace the value of current_directory from os.getcwd() to a path of your choice.

Writing user input to file-

```
121
122     def write_dict_to_file(params: dict, filename: str) -> None: 1 usage  ± thele +1
123         try:
124             with open(filename, "w") as file:
125                 for key, value in params.items():
126                     file.write(f"{key}:{value}\n")
127         except Exception as e:
128             print(f"Error while writing to file: {e}")
129
```

Function that gets a dictionary and a file name and writes the dictionary to this file.

Function to send the params to server and client-

```
113
114     def get_client_params(): 1 usage  ± thele
115         client_keys = ["message", "timeout", "window_size"]
116         return slice_json(client_keys)
117
118
119
120
121     def get_server_params(): 1 usage  ± thele *
122         server_keys = ["maximum_msg_size"]
123         params = slice_json(server_keys)
124         params["maximum_msg_size"] = int(params.get("maximum_msg_size", 0))
125         if params["maximum_msg_size"] < 4:
126             new_value = int(input("maximum_msg_size must be at least 4: enter new max size: "))
127             if new_value < 4:
128                 raise ValueError("maximum_msg_size must be at least 4")
129             params["maximum_msg_size"] = new_value
130
131     return params
132
```

Function returns to the server and the client the values each of them needs. The minimal size of maximum_msg_size is 4, function handle the case of invalid value.

Choosing a txt file-

```
103
104     def choose_text_file(text_files : list[str]) -> str: 1 usage  ↲ thele
105         print(f"found more than one text file: {text_files}")
106         path_to_params = int(input("Choose a text file: "))
107         return text_files[path_to_params]
108
```

Function that handles the file choice if there's multiple txt files to choose from.

Wireshark

First Run-

Mimic lost packages by not sending those packages at all (as seen in the client code description above). The server will not receive the lost packages while the client will treat them as sent (stores them in current packages list).

What we expect to see?

- 1- Client won't send the package but will store it as sent.
- 2- Server won't receive the lost package and will stop sending acks for the next packages it will receive.
- 3- One of the threshold will pass (time or sequence) and the client will find the lost package and resend it.
- 4- The server will receive the lost package and will send an ack for it, than- continue to send acks for the rest of the packages it received and didn't acknowledge yet.
- 5- Client will receive those acks, and return to sending the rest of the packages.

Packages To Lose = [4,9,10]

Txt file =

```
message: "aaaaabbbbcccccdddddeeeeeffffggggghhhiiiijjjjkkkklllll"
maximum_msg_size:4
window_size:3
timeout:2
```

Wireshark-

Key: 89, Payload (hex):
4d534700000000000061616262000020000000000003a4aa57226dfd9410000000020000002000000
| sequence: 2 | data: aabb | sent time: 1736219082.5826554 |ACK: False | prev seq: 2 | pos: 2 | header: MSG |

Key: 89-2, Payload (hex):
41434b000000000000000320000000000000000000000000000955cca7226dfd9410000000000000000000000000000000
| sequence: 0 | data: 2 | sent time: 1736219083.1619008 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 133, Payload (hex):
4d5347000000000000626262630000300000000000000b376d97226dfd941000000030000003000000
| sequence: 3 | data: bbbc | sent time: 1736219083.3978698 |ACK: False | prev seq: 3 | pos: 3 | header: MSG |

Key: 133-2, Payload (hex):
41434b0000000000000003300000000000000000000000000001d4cf7226dfd9410000000000000000000000000000000
| sequence: 0 | data: 3 | sent time: 1736219083.9577706 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Here is lost package 4 (which we didn't send), packages 5, 6 didn't receive an ack since package 4 didn't arrived to the server:

Key: 177, Payload (hex):
4d534700000000000064646464000050000000000005aa6427326dfd94100000000500000005000000
| sequence: 5 | data: dddd | sent time: 1736219085.0414033 |ACK: False | prev seq: 5 | pos: 5 | header: MSG |

Key: 221, Payload (hex):
4d534700000000000064656565000060000000000085e7767326dfd9410000000060000006000000
| sequence: 6 | data: deee | sent time: 1736219085.8578808 |ACK: False | prev seq: 6 | pos: 6 | header: MSG |

Reached seq threshold (3), resending of package 4:

Key: 265, Payload (hex):
4d534700000000000063636363000040000000000000837ccb7326dfd9410000000040000004000000
| sequence: 4 | data: cccc | sent time: 1736219087.1794746 |ACK: False | prev seq: 4 | pos: 4 | header: MSG |

Ack for package 4:

Key: 177-2, Payload (hex):
41434b00000000000000034000000000000000000000000000002e87cd7326dfd9410000000000000000000000000000000
| sequence: 0 | data: 4 | sent time: 1736219087.2113757 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Acks for packages 5,6 :

Key: 221-2, Payload (hex):
41434b00000000000035000000000000000000000005e8fcf7326dfd9410000000000000000000000000000000
| sequence: 0 | data: 5 | sent time: 1736219087.2431254 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 265-2, Payload (hex):
41434b00000000000036000000000000000000000004ba0d27326dfd9410000000000000000000000000000000
| sequence: 0 | data: 6 | sent time: 1736219087.2910335 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

continue to send data:

Key: 309, Payload (hex):
4d534700000000000065656660000700000000000000000595df7326dfd9410000000000000000000000000000000
| sequence: 7 | data: eeff | sent time: 1736219087.4934895 |ACK: False | prev seq: 7 | pos: 7 | header: MSG |

Key: 309-2, Payload (hex):
41434b0000000000003700000000000000000000000000000096047426dfd9410000000000000000000000000000000
| sequence: 8 | data: ffcc | sent time: 1736219087.5588119 |ACK: False | prev seq: 8 | pos: 8 | header: MSG |

| sequence: 0 | data: 7 | sent time: 1736219088.0716553 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |
Key: 353, Payload (hex):
4d53470000000000000066666667000008000000000000001b75147426dfd94100000000080000008000000

| sequence: 8 | data: fffg | sent time: 1736219088.3196476 |ACK: False | prev seq: 8 | pos: 8 | header: MSG |
Key: 353-2, Payload (hex):
41434b000000000000003800000000000000000000000005bc1357426dfd941000000000000000000000000000000

| sequence: 0 | data: 8 | sent time: 1736219088.8399265 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Lost packages 9, 10:

Key: 397, Payload (hex):
4d5347000000000000006869696900000b0000000000000000295bb17426dfd941000000000b000000b000000

| sequence: 11 | data: hiii | sent time: 1736219090.771189 |ACK: False | prev seq: 11 | pos: 11 | header: MSG |

Reached seq threshold (3), resending of package 9:

Key: 441, Payload (hex):
4d5347000000000000006767676700000900000000000000072ef057526dfd941000000000900000009000000

| sequence: 9 | data: gggg | sent time: 1736219092.0927396 |ACK: False | prev seq: 9 | pos: 9 | header: MSG |

Ack for package 9:

Key: 397-2, Payload (hex):
41434b0000000000000039000000000000000000000000000000daed077526dfd9410000000000000000000000000

| sequence: 0 | data: 9 | sent time: 1736219092.1238923 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Seq threshold is now 9 +3, continue to send the next package... (this approves that the window size was updated as expected after we received the ack)

Key: 485, Payload (hex):
4d53470000000000000069696a6a00000c0000000000000d3e4197526dfd941000000000c000000c000000

| sequence: 12 | data: iiji | sent time: 1736219092.4045913 |ACK: False | prev seq: 12 | pos: 12 | header: MSG |

Reached seq threshold (3), resending of package 10:

Key: 529, Payload (hex):
4d5347000000000000006868686800000a000000000000002ecc6e7526dfd94100000000a000000a000000

| sequence: 10 | data: hhhh | sent time: 1736219093.7312121 |ACK: False | prev seq: 10 | pos: 10 | header: MSG |

Ack for package 10:

Key: 441-2, Payload (hex):
41434b00000000000000313000

| sequence: 0 | data: 10 | sent time: 1736219093.7459815 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Acks for packages 11,12:

Key: 485-2, Payload (hex):
41434b000000000000003131000

| sequence: 0 | data: 11 | sent time: 1736219093.7919788 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 529-2, Payload (hex):
41434b00000000000000313200

| sequence: 0 | data: 12 | sent time: 1736219093.8079453 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

continue to send data:

Key: 573, Payload (hex):
 4d53470000000000000006a6a6a6b00000d0000000000000e094827526dfd941000000000d000000d00000
 | sequence: 13 | data: jjjk | sent time: 1736219094.0403366 |ACK: False | prev seq: 13 | pos: 13 | header: MSG |

Key: 573-2, Payload (hex):
 41434b0000000000000003133000000000000000000000ba24a47526dfd941000000000000000000000000000000
 | sequence: 0 | data: 13 | sent time: 1736219094.5647416 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 617, Payload (hex):
 4d534700000000000006b6b6b6b00000e0000000000000494cb77526dfd941000000000e0000000e000000
 | sequence: 14 | data: kkkk | sent time: 1736219094.864031 |ACK: False | prev seq: 14 | pos: 14 | header: MSG |

Key: 617-2, Payload (hex):
 41434b00000000000003134000000000000000000000000000dcc4d87526dfd9410000000000000000000000000000
 | sequence: 0 | data: 14 | sent time: 1736219095.3870153 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 661, Payload (hex):
 4d534700000000000006c6c6c6c00000f0000000000000f5e3eb7526dfd94100000000f0000000f000000
 | sequence: 15 | data: llll | sent time: 1736219095.6857884 |ACK: False | prev seq: 15 | pos: 15 | header: MSG |

Key: 661-2, Payload (hex):
 41434b000000000000031350000000000000000000000000058da0d7626dfd94100000000000000000000000000000
 | sequence: 0 | data: 15 | sent time: 1736219096.2164516 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 705, Payload (hex):
 4d534700000000000006c2200000001000000000000002a97207626dfd941000000010000001000000
 | sequence: 16 | data: l" | sent time: 1736219096.5092263 |ACK: False | prev seq: 16 | pos: 16 | header: MSG |

Key: 705-2, Payload (hex):
 41434b000000000000031360000000000000000000000000000d31b427626dfd94100000000000000000000000000
 | sequence: 0 | data: 16 | sent time: 1736219097.0329483 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 749, Payload (hex):
 444f4e4500000000000454f4d730000110000000000aa69617626dfd941000000011000001100000

Done package:

| sequence: 17 | data: EOMs | sent time: 1736219097.5220742 |ACK: False | prev seq: 17 | pos: 17 | header: DONE |

Key: 749-2, Payload (hex):
 41434b000000000000031370000000000000000000000000000d172637626dfd94100000000000000000000000000000
 | sequence: 0 | data: 17 | sent time: 1736219097.5538828 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 793, Payload (hex):
 434c4f5345000000000072657175000120000000000021a96e7626dfd941000000012000001200000

Closing packages:

| sequence: 18 | data: requ | sent time: 1736219097.7290728 |ACK: False | prev seq: 18 | pos: 18 | header: CLOSE |

Key: 793-2, Payload (hex):
 444953434f4e4e454354617070720000000000000000000000b28c727626dfd94100000000000000000000000000000000000000
 | sequence: 0 | data: appr | sent time: 1736219097.7898374 |ACK: False | prev seq: 0 | pos: 0 | header: DISCONNECT |

As seen, our code acted as we expected.

Second Run- mimic lost acks by not sending an ack for those packages (as seen in the server code description above). The client

What we expect to see?

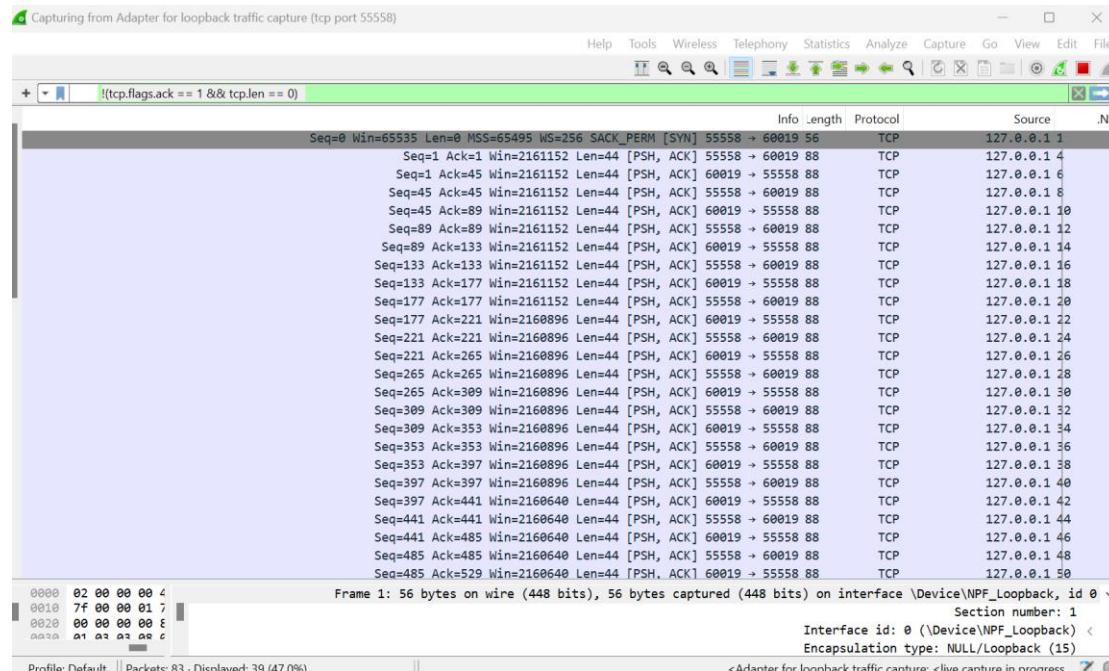
- 1- Client will send the package to the server.
- 2- Server will receive the package but won't send an ack for it (mimic the lost ack)
- 3- Client won't receive an ack for this package. Instead it will receive an ack for the next seq package that has been sent.
- 4- After getting the next ack, the client will know that all the packages prior to the ack number were received, and will mark them as acked.
- 5- Client continue to send the rest of the packages.

ACKs To Lose = [4,9,10]

Txt file =

```
1 message:"aaaaabbbbbccccccddddddeeeeffffffggggghhhhiiiijjjjkkkklllll"
2 maximum_msg_size:4
3 window_size:3
4 timeout:2
5
```

Wireshark-



Key: 265, Payload (hex):
4d53470000000000000006465656500000600000000000004d5c9f0e56dfd941000000006000000600000

| sequence: 6 | data: deee | sent time: 1736267834.4900086 |ACK: False | prev seq: 6 | pos: 6 | header: MSG |

Key: 221-2, Payload (hex):
41434b00000000000000360000000000000000000000001a7c20e56dfd9410000000000000000000000000000

| sequence: 0 | data: 6 | sent time: 1736267835.0415003 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 309, Payload (hex):
4d534700000000000656566660000700000000000000bdc1d30e56dfd9410000000070000007000000

Sending of package 7 approves that the client treats package 4 as acked. Else- seq threshold would stop the sending (like seen in previous example)

| sequence: 7 | data: eeff | sent time: 1736267835.3086998 |ACK: False | prev seq: 7 | pos: 7 | header: MSG |

Key: 265-2, Payload (hex):
41434b000000000000037000000000000000000000000002264f60e56dfd941000000000000000000000000000000

| sequence: 0 | data: 7 | sent time: 1736267835.8498616 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 353, Payload (hex):
4d53470000000000066666670000800000000000b08e080f56dfd941000000080000008000000

| sequence: 8 | data: fffg | sent time: 1736267836.133709 |ACK: False | prev seq: 8 | pos: 8 | header: MSG |

Key: 309-2, Payload (hex):
41434b00000000000003800000000000000000000334d2c0f56dfd941000000000000000000000000000000

| sequence: 0 | data: 8 | sent time: 1736267836.6922119 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Sending of package 9:

Key: 397, Payload (hex):
4d53470000000000067676767000090000000000010993c0f56dfd9410000000090000009000000

| sequence: 9 | data: gggg | sent time: 1736267836.9468422 |ACK: False | prev seq: 9 | pos: 9 | header: MSG |

No ack received for package 9:

Sending of package 10:

Key: 441, Payload (hex):
4d534700000000000686868680000a00000000006c61700f56dfd94100000000a000000a000000

| sequence: 10 | data: hhhh | sent time: 1736267837.7559462 |ACK: False | prev seq: 10 | pos: 10 | header: MSG |

No ack received for package 10:

Key: 485, Payload (hex):
4d534700000000000686969690000b00000000007798a40f56dfd94100000000b000000b000000

| sequence: 11 | data: hiii | sent time: 1736267838.5718057 |ACK: False | prev seq: 11 | pos: 11 | header: MSG |

Ack for package 11 was sent, client will consider package 9,10 as acked:

Key: 353-2, Payload (hex):
41434b00000000000313100000000000af35c60f56dfd941000000000000000000000000000000

| sequence: 0 | data: 11 | sent time: 1736267839.0970266 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Sending of package 12 approves that the client treats package 9 as acked. Else- seq threshold would stop the sending.

Key: 529, Payload (hex):
4d534700000000000069696a6a00000c000000000000000006394d80f56dfd941000000000c0000000c000000
| sequence: 12 | data: iijj | sent time: 1736267839.3840568 |ACK: False | prev seq: 12 | pos: 12 | header: MSG |
Key: 397-2, Payload (hex):
41434b00000000000000003132000000000000000000000000000000c0d8fa0f56dfd9410000000000000000000000000000000
| sequence: 0 | data: 12 | sent time: 1736267839.9194794 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Sending of package 13 approves that the client treats package 10 as acked. Else- seq threshold would stop the sending.

Key: 573, Payload (hex):
4d53470000000000006a6a6b00000d00000000000000060230c1056dfd941000000000d0000000d000000
| sequence: 13 | data: jjjk | sent time: 1736267840.189659 |ACK: False | prev seq: 13 | pos: 13 | header: MSG |
Key: 441-2, Payload (hex):
41434b0000000000000000313300000000000000000000009eb42e1056dfd9410000000000000000000000000000000
| sequence: 0 | data: 13 | sent time: 1736267840.729774 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 617, Payload (hex):
4d53470000000000006b6b6b6b00000e00000000000000f8d1401056dfd941000000000e000000e000000
| sequence: 14 | data: kkkk | sent time: 1736267841.0128155 |ACK: False | prev seq: 14 | pos: 14 | header: MSG |
Key: 485-2, Payload (hex):
41434b000000000000003134000000000000000000003248631056dfd94100000000000000000000000000000
| sequence: 0 | data: 14 | sent time: 1736267841.5512815 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 661, Payload (hex):
4d53470000000000006c6c6c00000f0000000000000748a741056dfd941000000000f000000f000000
| sequence: 15 | data: llll | sent time: 1736267841.8209505 |ACK: False | prev seq: 15 | pos: 15 | header: MSG |
Key: 529-2, Payload (hex):
41434b00000000000000003135000000000000000007bdf961056dfd9410000000000000000000000000000000
| sequence: 0 | data: 15 | sent time: 1736267842.3573902 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Key: 705, Payload (hex):
4d53470000000000006c220000000100000000000005c2fa81056dfd94100000000100000001000000
| sequence: 16 | data: l" | sent time: 1736267842.6278906 |ACK: False | prev seq: 16 | pos: 16 | header: MSG |
Key: 573-2, Payload (hex):
41434b00000000000000313600000000000000000fbfb9c91056dfd9410000000000000000000000000
| sequence: 0 | data: 16 | sent time: 1736267843.151962 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Done package:

Key: 749, Payload (hex):
444f4e45000000000000454f4d73000011000000000000e30ae91056dfd941000000001100000011000000
| sequence: 17 | data: EOMs | sent time: 1736267843.6412895 |ACK: False | prev seq: 17 | pos: 17 | header: DONE |
Key: 617-2, Payload (hex):
41434b0000000000000031370000000000000000000d12eb1056dfd941000000000000000000000000000000
| sequence: 0 | data: 17 | sent time: 1736267843.6729805 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Closing packages:

Key: 793, Payload (hex):

434c4f5345000000000007265717500012000000000000e549f61056dfd941000000012000001200000

| sequence: 18 | data: requ | sent time: 1736267843.8482602 |ACK: False | prev seq: 18 | pos: 18 | header: CLOSE |

Key: 661-2, Payload (hex):

444953434f4e4e454354617070720000000000000000005733f81056dfd941000000000000000000000000000000

| sequence: 0 | data: appr | sent time: 1736267843.8781335 |ACK: False | prev seq: 0 | pos: 0 | header: DISCONNECT |

Parts of the output of client and server code to analyze the behavior better:

Receiving of ack 3 – client output-

```
received ACK 3 !  
  
updated window size:  
CURRENT PACKAGES: ['\n| sequence: 1 | data: "aaa" | sent time: 1736269678.996142  
starting time window based on current time  
Updating SEQ window size by last acked pack: 3  
New seq window: 6 Prev seq window : 5
```

We can see that after the client receives ack 3, it updates window sizes. Since packages 1,2,3 are acked and we didn't send package 4 yet- time window will be updated according to the current time. Seq window will be updated according to the last acked package – package 3.

Sending package 4- client output-

```
Time window: 1736269683.171224 Seq window: 6  
SENDING package :  
  
| sequence: 4 | data: cccc | sent time: 1736269681.4558926 |ACK: False | prev seq: 4 | pos: 4 | header: MSG |  
added package to current packages:  
| sequence: 4 | data: cccc | sent time: 1736269681.4558926 |ACK: False | prev seq: 4 | pos: 4 | header: MSG |cu
```

The client checks the window before sending the package, and adds it to the sent packages list as expected. (note that I added a time.sleep() for the testing that caused the time window to be inaccurate- it will be removed from the submitted code, and from the testing of time threshold).

Receiving package 4- server output-

```
received package:  
| sequence: 4 | data: cccc | sent time: 1736269290.6009762 |ACK: False | prev seq: 4 | pos: 4 | header: MSG | last_seq = 3  
losing ack for pack: 4  
updated last seq: 4  
acks to lose: [9, 10]
```

Server receives package 4 and doesn't sends an ack. Updating last seq to package 4 as expected.

Sending of package 5 – client output-

```
Time window: 1736270451.9652874 Seq window: 6
SENDING package :

| sequence: 5 | data: dddd | sent time: 1736270451.06319 |ACK: False | prev seq: 5 | pos: 5 | header: MSG |
added package to current packages:
| sequence: 5 | data: dddd | sent time: 1736270451.06319 |ACK: False | prev seq: 5 | pos: 5 | header: MSG |current packages state: ['\n']
```

Client checks threshold (that still updated by package 3 as expected) and sends package 5.

Receiving of package 5 and sending the ack- server output-

```
received package:
| sequence: 5 | data: dddd | sent time: 1736270451.06319 |ACK: False | prev seq: 5 | pos: 5 | header: MSG | last_seq = 4
ACK5 sent!
updated last seq: 5
```

Receiving of ack 5- client output-

```
ACKED PACKAGE OUT OF ORDER:
| sequence: 5 | data: dddd | sent time: 1736270451.06319 |ACK: False | prev seq: 5 | pos: 5 | header: MSG |, acks [4] where lost
Correcting lost ack 4

received ACK 5 !
```

Client receives ack 5, recognize that it's bigger than expected and mark package 4 as asked. Than- marks package 5 as acked.

Updating window size based on package 5:

```
starting time window based on current time
Updating SEQ window size by last acked pack: 5
New seq window: 8 Prev seq window : 6
```

Another example- when losing acks 9,10- client fix all of the lost acks:

```
ACKED PACKAGE OUT OF ORDER:
| sequence: 11 | data: hiii | sent time: 1736270455.94869 |ACK: False | prev seq: 11 | pos: 11 | header: MSG |, acks [9, 10] where lost
Correcting lost ack 9
Correcting lost ack 10

received ACK 11 !
```

Test 3:

Here I wanted to test a case where acks 9,10,11 are lost, and seq window is 3. This will result in resending of package 9.

Unfortunately, my code failed this test- resulted in multiple trying to send package 9.

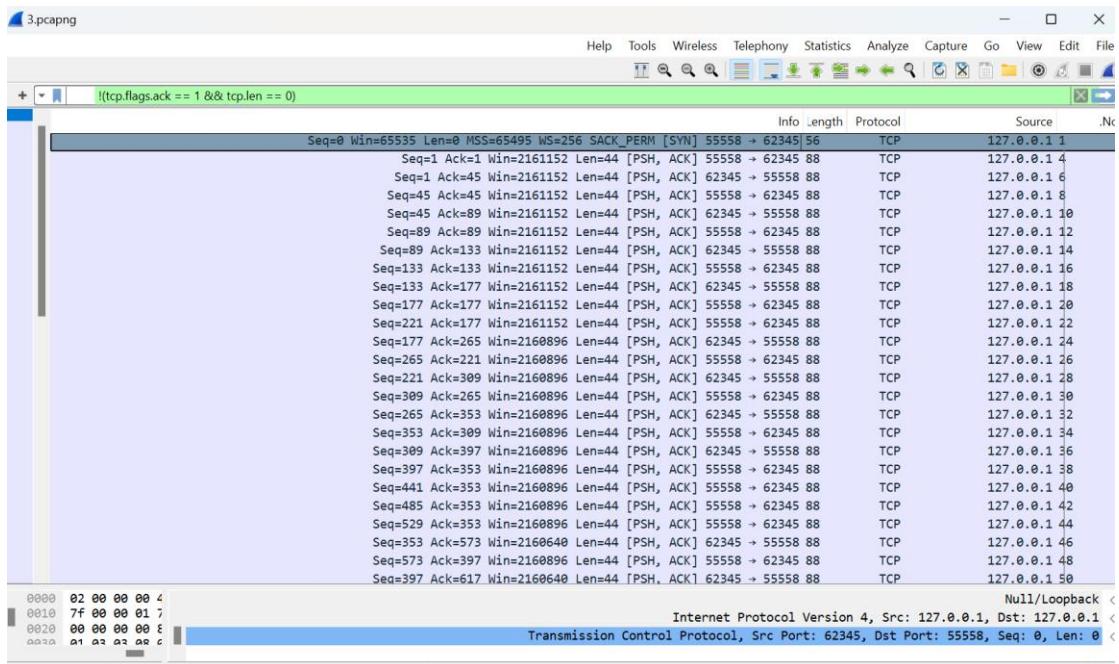
To handle this case, I modified server code as followed:

```
while msg_package.get_header() == "MSG":  
    try:  
        print(f"acks to lose: {ACKS_TO_LOSE}")  
        print(f"received package: {msg_package} last_seq = {last_seq}")  
        if int(msg_package.get_pos()) <= last_seq:  
            print(f"\nreceived duplicated package: resending an ack")  
            msg_package.send_ack(client_socket, PARAMS["maximum_msg_size"])  
        pass|
```

At the beginning of the receive msg loop, I added a test to check if ack is equal or smaller than previous ack, if so- server will send another ack for this package.

Now the code works as expected.

Wireshark-



Analyzing the JSON file:

I will include only the relevant parts from the output, since there's a lot of repeated data that was already included in the previous tests.

...Starting with the sending of package 8...

```
| sequence: 8 | data: ffff | sent time: 1736271844.804456 |ACK: False | prev seq: 8 | pos: 8 | header: MSG |
Key: 309-2, Payload (hex):
41434b00000000000000038000000000000000000000000bbe354f959dfd941000000000000000000000000000000
```

```
| sequence: 0 | data: 8 | sent time: 1736271845.3263996 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |
```

Sending packages 9,10,11 without receiving an ack:

```
Key: 397, Payload (hex):
4d5347000000000000000676767670000900000000000021e367f959dfd9410000000090000009000000
```

```
| sequence: 9 | data: gggg | sent time: 1736271845.6232378 |ACK: False | prev seq: 9 | pos: 9 | header: MSG |
Key: 441, Payload (hex):
4d53470000000000000686868680000a0000000000000ddf69cf959dfd94100000000a0000000a000000
```

```
| sequence: 10 | data: hhhh | sent time: 1736271846.4525673 |ACK: False | prev seq: 10 | pos: 10 | header: MSG |
Key: 485, Payload (hex):
4d53470000000000000686969690000b000000000000000f7fd0f959dfd94100000000b000000b000000
```

```
| sequence: 11 | data: hiii | sent time: 1736271847.257755 |ACK: False | prev seq: 11 | pos: 11 | header: MSG |
```

Resending package 9 since seq window has passed:

```
Key: 529, Payload (hex):
4d534700000000000006767676700009000000000000be5225fa59dfd9410000000090000009000000
```

```
| sequence: 9 | data: gggg | sent time: 1736271848.5831752 |ACK: False | prev seq: 9 | pos: 9 | header: MSG |
```

Server recognized this package was received for the second time and sends an ack:

```
Key: 353-2, Payload (hex):
41434b00000000000003900000000000000000000000e7672cfa59dfd9410000000000000000000000
```

```
| sequence: 0 | data: 9 | sent time: 1736271848.6938417 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |
```

Client sends package 12, because seq threshold increased:

```
Key: 573, Payload (hex):
4d534700000000000069696a6a0000c000000000000000000000000000000000000000000000000000000000
```

```
| sequence: 12 | data: ijjj | sent time: 1736271848.8985054 |ACK: False | prev seq: 12 | pos: 12 | header: MSG |
```

```
Key: 397-2, Payload (hex):
41434b00000000000003132000000000000000000000359b5cfa59dfd941000000000000000000000000000000
```

Server sends an ack for package 12:

```
| sequence: 0 | data: 12 | sent time: 1736271849.446973 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |
```

```
Key: 617, Payload (hex):
4d53470000000000006a6a6b0000d000000000000054cc6dfa59dfd94100000000d00000000000000000000
```

Client received the ack12, and consider packages 10,11 as acked. Continue transferring data:

```
| sequence: 13 | data: jjjk | sent time: 1736271849.7155962 |ACK: False | prev seq: 13 | pos: 13 | header: MSG |
```

.....

Analyzing server and client code output:

Client sends packages 9,10,11 without receiving an ack-

```

| sequence: 9 | data: gggg | sent time: 1736272844.7187493 |ACK: False | prev seq: 9 | pos: 9 | header: MSG |
added package to current packages:
| sequence: 9 | data: gggg | sent time: 1736272844.7187493 |ACK: False | prev seq: 9 | pos: 9 | header: MSG |current packages state: ['\n'
Time window: 1736272846.4961421 Seq window: 11
SENDING package :

| sequence: 10 | data: hhhh | sent time: 1736272845.5255713 |ACK: False | prev seq: 10 | pos: 10 | header: MSG |
added package to current packages:
| sequence: 10 | data: hhhh | sent time: 1736272845.5255713 |ACK: False | prev seq: 10 | pos: 10 | header: MSG |current packages state: ['\n'
Time window: 1736272846.4961421 Seq window: 11
SENDING package :

| sequence: 11 | data: hiii | sent time: 1736272846.3361585 |ACK: False | prev seq: 11 | pos: 11 | header: MSG |
added package to current packages:
| sequence: 11 | data: hiii | sent time: 1736272846.3361585 |ACK: False | prev seq: 11 | pos: 11 | header: MSG |current packages state: ['\n'

```

Server receives packages 9,10,11 and doesn't send the ack-

```

received package:
| sequence: 9 | data: gggg | sent time: 1736272844.7187493 |ACK: False | prev seq: 9 | pos: 9 | header: MSG | last_seq = 8
losing ack for pack: 9
updated last seq: 9
acks to lose: [10, 11]
received package:
| sequence: 10 | data: hhhh | sent time: 1736272845.5255713 |ACK: False | prev seq: 10 | pos: 10 | header: MSG | last_seq = 9
losing ack for pack: 10
updated last seq: 10
acks to lose: [11]
received package:
| sequence: 11 | data: hiii | sent time: 1736272846.3361585 |ACK: False | prev seq: 11 | pos: 11 | header: MSG | last_seq = 10
losing ack for pack: 11
updated last seq: 11
acks to lose: []

```

Clients time and seq threshold passed, resending lost package 9:

```

THRESHOLD PASSED (time):
Time Window: 1736272846.7187493, Current time: 1736272847.1427433
resend lost package
CURRENT PACKAGES: ['\n'] sequence: 1 | data: "aaa" | sent time: 1736272838.2149637 |ACK: True
min pos, no ack:
| sequence: 9 | data: gggg | sent time: 1736272844.7187493 |ACK: False | prev seq: 9 | pos:

transfer for resending package:
| sequence: 9 | data: gggg | sent time: 1736272844.7187493 |ACK: False | prev seq: 9 | pos:
Time wind: 1736272846.7187493 Seq win: 11
RESENDING package :

| sequence: 9 | data: gggg | sent time: 1736272844.7187493 |ACK: False | prev seq: 9 | pos:

```

Client waits to receive an ack before continuing the sending-

Time to wait = "timeout"

```

waiting for ack: 9
time left: 2.0

```

Server received package 9, recognized that it was already received and sends an ack-

```
received package:  
| sequence: 9 | data: gggg | sent time: 1736272847.6504097 |ACK: False |  
  
received duplicated package: resending an ack  
ACK9 sent!
```

Client received the ack for package 9 and updated window sizes:

```
received ACK 9 !  
  
updated window size:  
CURRENT PACKAGES: ['\n| sequence: 1 | data: "aaa | sent time: 1736272838.2149637 |AC  
min pos, no ack:  
| sequence: 10 | data: hhhh | sent time: 1736272845.5255713 |ACK: False | prev seq:  
  
Updating TIME window by package 10 :  
New time window 1736272847.5255713, Prev time window: 1736272846.7187493  
Updating SEQ window size by last acked pack: 9  
New seq window: 12 Prev seq window : 11
```

time window- according to package 10, which is the smallest packet that didn't receive an ack.

Seq window- by last acked package (9)

Client sends package 12:

```
Time window: 1736272847.5255713 Seq window: 12  
SENDING package :  
  
| sequence: 12 | data: iijj | sent time: 1736272847.965898 |ACK: False | prev seq: 12 | pos: 12 | header: MSG |
```

Client fixing the lost acks:

```
ACKED PACKAGE OUT OF ORDER:  
| sequence: 12 | data: iijj | sent time: 1736272847.965898 |ACK: False |  
Correcting lost ack 10  
Correcting lost ack 11  
  
received ACK 12 !
```

As seen, the code handles this case as expected.

Test 4:

Losing acks+packages that follows each other.

ACKS_TO_LOSE- 6,8,10

PACKGES_TO_LOSE-7,9,11

Analyzing JSON decode + output from code

```
Key: 221, Payload (hex):
4d5347000000000000646565656500050000000000000007f051d0b5cdfd941000000005000000500000

| sequence: 5 | data: deee | sent time: 1736273964.4534605 |ACK: False | prev seq: 5 | pos: 5 | header: MSG |

Key: 221-2, Payload (hex):
41434b0000000000000350000000000000000000002fc3f0b5cdfd941000000000000000000000000000000

| sequence: 0 | data: 5 | sent time: 1736273964.9997563 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |
```

JSON- package 6 sending-

```
Key: 265, Payload (hex):
4d5347000000000006566666660006000000000001f51510b5cdfd9410000000060000006000000

| sequence: 6 | data: efff | sent time: 1736273965.2705762 |ACK: False | prev
seq: 6 | pos: 6 | header: MSG |
```

Client output – package 6 sending-

```
SENDING package :

| sequence: 6 | data: effff | sent time: 1736274187.8852105 |ACK: False | prev seq: 6 | pos: 6 | header: MSG |
```

Server output- skipping ack for package 6- "ack 6 was lost"-

```
received package:
| sequence: 6 | data: effff | sent time: 1736274187.8852105 |ACK: False |
losing ack for pack: 6
updated last seq: 6
```

Client output – skipping sending package 7- "package 7 was lost"-

```
added package to current packages:
| sequence: 7 | data: fgggg | sent time: 1736274188.7016888 |ACK: False | prev seq: 7 | pos: 7 |
package 7 will be lost
```

JSON- package 8 sending-

```
Key: 309, Payload (hex):
4d534700000000000676868686000800000000000009f84b90b5cdfd9410000000080000008000000

| sequence: 8 | data: ghhh | sent time: 1736273966.8987195 |ACK: False | prev
seq: 8 | pos: 8 | header: MSG |
```

Client output – package 8 sending-

```
SENDING package :

| sequence: 8 | data: ghhh | sent time: 1736274189.5231082 |ACK: False | prev seq: 8 | pos: 8 | header: MSG |
```

Server output- skipping ack for package 8- "ack 8 was lost"-

```
received package:
| sequence: 8 | data: ghhh | sent time: 1736274189.5231082 |ACK: False | prev seq: 8 | pos: 8 | header: MSG | last_seq = 6
acks to lose: [8, 10]
```

JSON- resending of package 6-

Key: 353, Payload (hex):

```
4d5347000000000000656666666000600000000000000009dac0d0c5cdfd9410000000006000000060000  
00
```

```
| sequence: 6 | data: efff | sent time: 1736273968.2136605 |ACK: False | prev seq: 6 | pos: 6 | header: MSG |
```

Client output – resending of package 6-

```
RESENDING package :  
  
| sequence: 6 | data: efffff | sent time: 1736274187.8852105 |ACK: False | prev seq: 6 | pos:  
waiting for ack: 6  
time left: 2.0
```

JSON- ack 6-

Key: 265-2, Payload (hex):

```
41434b00000000000036000000000000000000000000000000c79f100c5cdfd941000000000000000000000000000000
```

```
| sequence: 0 | data: 6 | sent time: 1736273968.259752 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |
```

Server output – sending ack 6-

```
received package:  
| sequence: 6 | data: efffff | sent time: 1736274190.8450372 |ACK: Fal  
  
received duplicated package: resending an ack  
ACK6 sent!
```

Client output- after receiving ack6, updating windows-

```
Updating TIME window by package 7 :  
New time window 1736274190.7016888, Prev time window: 1736274189.8852105  
Updating SEQ window size by last acked pack: 6  
New seq window: 9 Prev seq window : 8
```

Than- "losing package 9"-

```
| sequence: 9 | data: hiii | sent time: 1736274191.1473224 |ACK: False | prev seq: 9 | pos: 9  
package 9 will be lost
```

JSON- resending of package 7-

Key: 397, Payload (hex):

```
4d5347000000000000666767676700070000000000000a282760c5cdfd9410000000070000  
0007000000
```

```
| sequence: 7 | data: fggg | sent time: 1736273969.8517232 |ACK: False | prev seq: 7 | pos: 7 | header: MSG |
```

[Client output- resending of package 7-](#)

```
THRESHOLD PASSED (time):
Time Window: 1736274190.7016888, Current time: 1736274191.9500923

transfer for resending package:
| sequence: 7 | data: fgggg | sent time: 1736274188.7016888 |ACK: False | prev seq: 7
```

[JSON- ack 7-](#)

Key: 309-2, Payload (hex):
41434b000000000000000370000000000000000000000000000199e7b0c5cdfd94100000000000000000000000

```
| sequence: 0 | data: 7 | sent time: 1736273969.9315245 |ACK: False | prev seq: 0
| pos: 0 | header: ACK |
```

[Server – ack- 7](#)

```
received package:
| sequence: 7 | data: fgggg | sent time: 1736274192.4592347 |ACK: False
ACK7 sent!
```

[JSON- sending package 10-](#)

Key: 441, Payload (hex):
4d534700000000000696a6a6a6a000a00000000000092d7890c5cdfd94100000000a000000a000000

```
| sequence: 10 | data: ijij | sent time: 1736273970.1537824 |ACK: False | prev seq: 10 | pos: 10 | header: MSG |
```

[Client – received ack 7 and updated thresholds-](#)

```
Updating TIME window by package 8 :
New time window 1736274191.5231082, Prev time window: 1736274190.7016888
Updating SEQ window size by last acked pack: 7
New seq window: 10 Prev seq window : 9
```

[Sending package 10 -](#)

```
SENDING package :

| sequence: 10 | data: ijijjj | sent time: 1736274192.7736154 |ACK: False | prev seq: 10 | pos: 10 | header: MSG |
```

[Resending package 8, since we reached threshold again:](#)

Key: 485, Payload (hex):
4d5347000000000006768686868000800000000000000c7dfde0c5cdfd94100000000080000
0008000000

```
| sequence: 8 | data: ghhh | sent time: 1736273971.4824083 |ACK: False | prev seq: 8 | pos: 8 | header: MSG |
```

Client code:

```
THRESHOLD PASSED (time):
```

```
Time Window: 1736274191.5231082, Current time: 1736274193.594702
```

```
RESENDING package :

| sequence: 8 | data: ghhh | sent time: 1736274189.5231082 |ACK: False | prev seq: 8 | pos: 8 | header: MSG |
waiting for ack: 8
time left: 2.0
```

Ack for package 8:

Key: 353-2, Payload (hex):

```
41434b0000000000000003800000000000000000000000000049dae20c5cdfd9410000000000000000000000000000
```

```
| sequence: 0 | data: 8 | sent time: 1736273971.544573 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |
```

Server output: recognized it's a package he already received and send another ack.

```
received package:
| sequence: 8 | data: ghhh | sent time: 1736274194.0986335 |ACK: False | prev seq: 8

received duplicated package: resending an ack
ACK8 sent!
```

Client output: starting with updating window size -

```
Updating TIME window by package 9 :
New time window 1736274193.1473224, Prev time window: 1736274191.5231082
Updating SEQ window size by last acked pack: 8
New seq window: 11 Prev seq window : 10
```

Now client can send package 11- but we decided to "lose" this package-

```
added package to current packages:
| sequence: 11 | data: jkkkk | sent time: 1736274194.400803 |ACK: False | prev seq: 11 | pos: 11
package 11 will be lost
```

Resending package 9, since we reached threshold again:

```
THRESHOLD PASSED (time):
```

```
Time Window: 1736274193.1473224, Current time: 1736274195.222952
```

Resending package 9 and sending ack9-

Key: 529, Payload (hex):

```
4d5347000000000006869696900090000000000000000385d460d5cdfd94100000000090000009000000
```

```
| sequence: 9 | data: hiii | sent time: 1736273973.0994396 |ACK: False | prev seq: 9 | pos: 9 | header: MSG |
```

Key: 397-2, Payload (hex):

```
41434b00000000000000003900000000000000000000000000000001e704b0d5cdfd9410000000000000000000000000000
```

```
| sequence: 0 | data: 9 | sent time: 1736273973.178718 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |
```

After receiving the ack- client updates the windows-

```
Updating TIME window by package 10 :  
New time window 1736274194.7736154, Prev time window: 1736274193.1473224  
Updating SEQ window size by last acked pack: 9  
New seq window: 12 Prev seq window : 11
```

Now, server should send ack for package 10, that he already received, but we decide to "lose" ack10-

```
Received package:  
| sequence: 9 | data: hiiii | sent time: 1736274195.727094 |ACK: False | prev seq: 9 | pos:  
ACK9 sent!  
updated last seq: 9  
Losing ack for pack: 10  
updated last seq: 10
```

Sending package 12-

Key: 573, Payload (hex):

```
4d53470000000000006b6c6c6c000c00000000000009191590d5cdfd94100000000c000000c000000
```

```
| sequence: 12 | data: klll | sent time: 1736273973.3995097 |ACK: False | prev seq: 12 | pos: 12 | header: MSG |
```

Server received package 12 but won't send an ack since package 10 is missing:

```
received package:  
| sequence: 12 | data: kllll | sent time: 1736274196.04153 |ACK: False | prev :  
-
```

Client threshold passed, resending lost package:

```
THRESHOLD PASSED (time):  
Time Window: 1736274194.7736154, Current time: 1736274196.8600402
```

Sending of package 10, ACK 10:

Key: 617, Payload (hex):

```
4d534700000000000696a6a6a6a000a000000000000021f7ae0d5cdfd94100000000a000000a000000
```

| sequence: 10 | data: ijij | sent time: 1736273974.7338336 |ACK: False | prev seq: 10 | pos: 10 | header: MSG |

Key: 441-2, Payload (hex):
41434b0000000000000031300000000000000000000de2b00d5cdfd94100000000000000000000000

| sequence: 0 | data: 10 | sent time: 1736273974.7638314 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Client output:

```
| sequence: 10 | data: ijijjj | sent time: 1736274192.7736154 |ACK: False | prev seq: 10 | pos: 10 | header: MSG |
waiting for ack: 10
time left: 2.0
received ACK 10 !
```

Server output- recognize the ack10 was lost and resending the ack-

```
received package:
| sequence: 10 | data: ijijjj | sent time: 1736274197.3652456 |ACK: False | prev seq: 10 | pos: 10 | he
received duplicated package: resending an ack
ACK10 sent!
```

Client updates windows after receiving ack10-

```
Updating TIME window by package 11 :
New time window 1736274196.400803, Prev time window: 1736274194.7736154
Updating SEQ window size by last acked pack: 10
New seq window: 13 Prev seq window : 12
```

Sending of package 13- threshold allows to send it

Key: 661, Payload (hex):
4d5347000000000006c2200000000d00000000000081e2c20d5cdfd94100000000d000000d000000

| sequence: 13 | data: l" | sent time: 1736273975.0450747 |ACK: False | prev seq: 13 | pos: 13 | header: MSG |

Server- received package 13, but won't send an ack since package 11 is lost-

```
received package:
| sequence: 13 | data: l" | sent time: 1736274197.6772366 |ACK: False | prev seq: 13 | pos
```

Sending of package 11-

Key: 705, Payload (hex):
4d5347000000000006a6b6b6b000b000000000000ce5f240e5cdfd94100000000b000000b000000

| sequence: 11 | data: jkkk | sent time: 1736273976.5683475 |ACK: False | prev seq: 11 | pos: 11 | header: MSG |

Client output- package 13 was the last package to send, so client moves to "handle lost packages before close". Anyway, even if it wasn't the last package- sequence window has passed and package 11 would have been sent again-

```
handling lost packages before close:  
CURRENT PACKAGES: ['\n| sequence: 1 | data: "aaaa | sent time: 1736274183.  
min pos, no ack:  
| sequence: 11 | data: jkkkk | sent time: 1736274194.400803 |ACK: False |
```

```
RESENDING package :  
  
| sequence: 11 | data: jkkkk | sent time: 1736274194.400803 |ACK: False  
waiting for ack: 11  
time left: 2.0  
received ACK 11 !
```

Sending of Ack 11:

Key: 485-2, Payload (hex):
41434b0000000000000003131000000000000000000000000002960290e5cdfd94100000000000000000000000000000000

```
| sequence: 0 | data: 11 | sent time: 1736273976.6464942 |ACK: False | prev seq:  
0 | pos: 0 | header: ACK |
```

Server output-

```
received package:  
| sequence: 11 | data: jkkkk | sent time: 1736274199.1925619 |ACK: False | prev seq:  
ACK11 sent!  
updated last seq: 11
```

Now, the server can send all the previous acks for packages 12,13-

Key: 529-2, Payload (hex):
41434b000000000000000313200000000000000000000000000094652d0e5cdfd941000000000000000000000000000000

```
| sequence: 0 | data: 12 | sent time: 1736273976.7093248 |ACK: False | prev seq:  
0 | pos: 0 | header: ACK |
```

Key: 573-2, Payload (hex):
41434b0000000000000003133000000000000000000000000000b9722f0e5cdfd941000000000000000000000000000000

```
| sequence: 0 | data: 13 | sent time: 1736273976.741377 |ACK: False | prev seq: 0  
| pos: 0 | header: ACK |
```

Server output-

```
ACK12 sent!  
updated last seq: 12  
ACK13 sent!  
updated last seq: 13
```

Sending end of message package- to indicate the server that the message is complete-

Key: 749, Payload (hex):

444f4e45000000000000454f4d7367000e00000000000005576380e5cdfd94100000000e0000000e0000000

| sequence: 14 | data: EOMs | sent time: 1736273976.8822224 |ACK: False | prev seq: 14 | pos: 14 | header: DONE |

Client output- calls for handle lost packages before close, to make sure all data was sent before sending the DONE package. As seen, handle lost packages didn't found any lost packages, so the client sends DONE package-

```
handling lost packages before close:  
finish current transfer:  
CURRENT PACKAGES: ['\n| sequence: 1 | data: "aaaa | sent time: 1736274183.8213613 |ACK: True | p  
min pos, no ack:  
| sequence: 14 | data: EOMsg | sent time: 1736274199.4998214 |ACK: False | prev seq: 14 | pos: 1  
  
Time left: 2.0 seconds
```

Ack 14-

Key: 617-2, Payload (hex):

41434b000000000000000313400000000000000000000000000000246d390e5cdfd94100000000000000000000000000000

| sequence: 0 | data: 14 | sent time: 1736273976.8972864 |ACK: False | prev seq: 0 | pos: 0 | header: ACK |

Server sends the client ack to indicate he received the "DONE" package.

```
done sending file  
warning! lost data: ['effff', 'ghhhh', 'ijjjj']  
ACK14 sent!  
msg packages: ['"aaaa', 'abbbb', 'bcccc', 'cdsss', 'deeee', 'effff', 'fgggg', 'ghhhh', 'hiiii', 'ijjjj', 'jkkkk', 'klfff', 'l"']  
  
full message received:  
"aaaabbcccdddeeeeefffffgggghhhhiiiijjjjkkkkllll"
```

The server rebuilds the full message. The warning is to warn if there's possible data lost. Here we can see that those are the duplicated packages that has been sent due to "lost acks".

Client receives ack 14:

```
received ACK 14 !
```

Sending CLOSE package and receiving DISCONNECT response from the server, to approve the client he can disconnect-

Key: 793, Payload (hex):

434c4f5345000000000007265717565000f000000000000666c450e5cdfd94100000000f000000f000000

```
| sequence: 15 | data: requ | sent time: 1736273977.084741 |ACK: False | prev seq: 15 | pos: 15 | header: CLOSE |
```

Key: 661-2, Payload (hex):
444953434f4e4e454354617070726f0000000000000000436d480e5cdfd94100000000000000000000000000000000

```
| sequence: 0 | data: appr | sent time: 1736273977.1316688 |ACK: False | prev seq: 0 | pos: 0 | header: DISCONNECT |
```

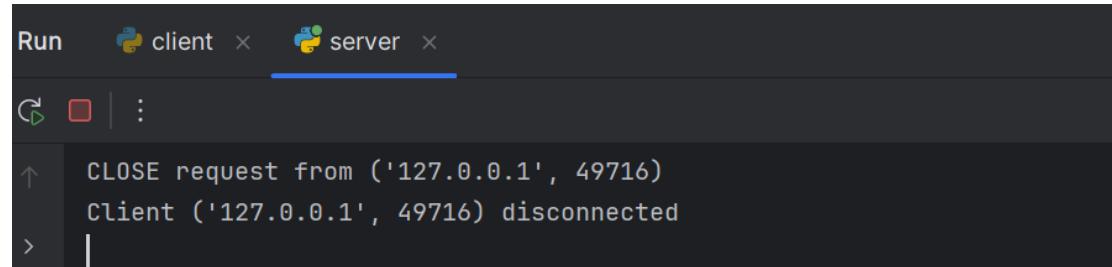
[Client output after receiving DISCONNECT package from the server-](#)

```
Received DISCONNECT msg from server:  
| sequence: 0 | data: appro | sent time: 1736279553.8857515 |ACK: False | prev seq: 0 | pos: 0 | header: DISCONNECT |  
Closing connection...
```

[Prints all the packages that were sent \(for debugging\) and the close the program-](#)

```
sequence: 14 | data: E0Msg | sent time: 1736279553.6521187 |ACK: True  
  
sequence: 15 | data: request to close connection | sent time: 17362795  
  
Process finished with exit code 0
```

[Server output after receiving CLOSE package from the client-](#)



```
Run   Python client x   Python server x
      | :  
↑  CLOSE request from ('127.0.0.1', 49716)  
>  Client ('127.0.0.1', 49716) disconnected
```

[Server stays open and can accept new clients.](#)

Test 5:

Lets try to lose DONE message.

Modified the server code –

Global parameter-

```

33     DONE_COUNT = 2
34

if msg_package.get_header() == "DONE":
    if DONE_COUNT == 0:
        print(f"done sending file")
        if msg_list or len(msg_list) > 0:
            print(f"warning! lost data: {[pack.get_payload() for pack in msg_list]}")
            MSG_DONE_Header(client_socket, msg_package, full_msg)
    else:
        print("skipping DONE...")
        DONE_COUNT -=1
        msg_package = Package(header: "MSG", payload: "")
        MSG_Header(client_socket, msg_package, full_msg =full_msg)

```

This will cause the server to pretend he lost DONE message twice (calls msg_header with empty data and the msg_list in order to mimic staying inside the receive msg loop).

Changed text file since there's no need to split a lot of packages for this test.

```

message: "aaaaabbbbbcccccddddddeeeeefffffgggggghhhhhiiiii jjjjjkkkkkllll"
maximum_msg_size:10
window_size:3
timeout:1

```

Analyzing Json –

#1 DONE message sent

Key: 353, Payload (hex):

444f4e450000000000000454f4d7367000000000008000000e965537863dfd941000000000800000008000000

| sequence: 0 | data: EOMs | sent time: 1736281569.303095 |ACK: False | prev seq: 8 | pos: 8 | header: DONE |

Ignore all ack 0, they created because of the way I forced the server to return to the while header == msg. if a real lost of DONE package would accrue, the server won't break outside of this loop in the first place.

Key: 353-2, Payload (hex):

41434b0000000000000030000000000000000000000000000002f7c597863dfd941000000000100000001000000

| sequence: 0 | data: 0 | sent time: 1736281569.3982046 |ACK: False | prev seq: 1 | pos: 1 | header: ACK |

#2 DONE message sent

Key: 397, Payload (hex):

444f4e450000000000000454f4d736700000000000800000e965537863dfd9410000000080000008000000

| sequence: 0 | data: EOMs | sent time: 1736281569.303095 |ACK: False | prev seq: 8 | pos: 8 | header: DONE |

Key: 397-2, Payload (hex):

41434b0000000000000003100000000000000000000000bb2a97863dfd9410000000020000002000000

| sequence: 0 | data: 1 | sent time: 1736281570.6514919 |ACK: False | prev seq: 2 | pos: 2 | header: ACK |

#3 DONE message sent- this time server will send an ack

Key: 441, Payload (hex):

444f4e45000000000000454f4d73670000000000800000e965537863dfd9410000000080000008000000

| sequence: 0 | data: EOMs | sent time: 1736281569.303095 |ACK: False | prev seq: 8 | pos: 8 | header: DONE |

DONE ack (seq8)

Key: 441-2, Payload (hex):

41434b00000000000000038000000000000000000000000000009931f37863dfd9410000000020000002000000

| sequence: 0 | data: 8 | sent time: 1736281571.7999022 |ACK: False | prev seq: 2 | pos: 2 | header: ACK |

Closing messages:

Key: 485, Payload (hex):

434c4f534500000000007265717565737420746f09000004953fe7863dfd9410000000090000009000000

| sequence: 1869881460 | data: requ | sent time: 1736281571.9738333 |ACK: False | prev seq: 9 | pos: 9 | header: CLOSE |

Key: 485-2, Payload (hex):

444953434f4e4e454354617070726f76696e672000000005668017963dfd9410000000020000002000000

| sequence: 543649385 | data: appr | sent time: 1736281572.0219932 |ACK: False | prev seq: 2 | pos: 2 | header: DISCONNECT |

And from server output we can see the full msg is received correctly-

```
ACK8 sent!
msg packages: [', ''aaaaabbbb', ', 'bcccccddd', 'deeeeeffff', 'fggggghhhh', 'hiiiiijjjj', 'jkkkkkllll', 'l''']

full message received:
"aaaaabbbbcccccdddeeeeffffgggghhhhiiiijjjjkkkkllll"
```

Sources-

ChatGPT

Important note- uploaded my code to the chat, this is not a code written by ChatGPT- those are his suggestions to modify the code.

First prompt-

Uploaded my code with a request for debugging-

"server doesn't print the data but does send it back correctly, find reasons that can cause this issue"

Chat response-

If the server sends the data back correctly but doesn't print it, the issue is likely within the handle_data function on the server. Here's how you can debug and fix it:

Possible Causes

1. **Data Encoding Issue:** The data received might be in bytes, and the print(data) in handle_data might not display the expected output.
 2. **handle_data Not Being Called Properly:** If handle_data is skipped or improperly implemented, it won't print the data.
 3. **Buffering or Partial Data:** The data sent from the client might not be received fully or correctly due to buffering or size mismatches.
-

Debugging Steps

1. **Add Debug Statements in handle_client:** Modify handle_client to confirm the received data before passing it to handle_data:

python

Copy code

```
def handle_client(CLIENT_SOCKET, client_address):  
    while True:  
        try:  
            data = CLIENT_SOCKET.recv(BUFSIZ)
```

```
if not data:  
    break  
  
    print(f"Raw data received: {data}") # Debug statement  
    handle_data(data, CLIENT_SOCKET, client_address)  
  
except OSError as e:  
    print(e)  
    break  
  
finally:  
    CLIENT_SOCKET.close()
```

2. **Check the Implementation of handle_data:** Verify the function correctly prints the decoded data:

python

Copy code

```
def handle_data(data, CLIENT_SOCKET, client_address):  
    decoded_data = data.decode() # Decode the data  
    print(f"Processed data from ({client_address[0]}:{client_address[1]}):  
{decoded_data}")  
    CLIENT_SOCKET.send(f"Server received: {decoded_data}".encode()) # Send response
```

3. **Inspect Client's Sent Data:** Ensure the client is sending data correctly as bytes:

python

Copy code

```
send_data(msg.encode(), client_socket) # Ensure encoding before sending
```

4. **Test with Simplified Logic:** Temporarily simplify the server to isolate the issue:

python

Copy code

```
def handle_client(CLIENT_SOCKET, client_address):  
    while True:  
        data = CLIENT_SOCKET.recv(BUFSIZ)  
        if not data:  
            break  
        print(f"Received from {client_address}: {data.decode()}")  
        CLIENT_SOCKET.send(b"Echo: " + data)
```

Adjusted Server Code

Here's a corrected version of the relevant server logic:

python

Copy code

```
def handle_client(CLIENT_SOCKET, client_address):  
    print(f"Connection established with {client_address}")  
    while True:  
        try:  
            data = CLIENT_SOCKET.recv(BUFSIZ)  
            if not data:  
                print(f"Client {client_address} disconnected")  
                break  
            print(f"Raw data received from {client_address}: {data}")  
            handle_data(data, CLIENT_SOCKET, client_address)  
        except OSError as e:  
            print(f"Error with client {client_address}: {e}")  
            break  
    finally:  
        CLIENT_SOCKET.close()
```

```
def handle_data(data, CLIENT_SOCKET, client_address):  
    decoded_data = data.decode()  
  
    print(f"Processed data from {client_address}: {decoded_data}")  
  
    response = f"Server received: {decoded_data}"  
  
    CLIENT_SOCKET.send(response.encode())
```

Verify the Fix

1. **Run the server and observe if it prints the received data.**
 2. **Check the client behavior to ensure it receives the server's response correctly.**
-

If the issue persists, provide additional context, such as the exact data being sent or any error logs. This will help refine the debugging process!

Chat response wasn't helpful at all, second prompt-

"prints doesn't works at all"

ChatGPT said:

ChatGPT

If the print statements in your server code are not working, it indicates one of the following issues:

Possible Causes

1. **Output is Buffered:** Python's print function buffers output, and the prints may not appear immediately in some environments.
2. **Server Code is Not Reaching the print Statements:** The relevant part of the code may not be executed due to errors or incorrect logic.
3. **Server Script is Running in a Suppressed Output Environment:** If you're running the server in an IDE or terminal with suppressed output, the prints may not appear.
4. **Threading Issue:** If the print is inside a thread and the thread is not running correctly, the code with print will not execute.

Gave some examples I didn't used, but I realized that the issue is probably with one of the threads.

Prompt 3-

"prints still doesn't work, what another issue with threads can cause this?"

Thought about thread handling issues for 15 seconds

Below are some **common threading pitfalls** (beyond removing client_THREAD.join() in your accept_incoming_connections loop) that can cause your server to exit early or fail to show print statements. Go through each to ensure none are happening in your code or environment.

1. Calling the Function Instead of Passing It to Thread

A frequent mistake is writing:

python

Copy code

```
ACCEPT_THREAD =  
    Thread(target=accept_incoming_connections(server_socket))
```

Instead of:

python

Copy code

```
ACCEPT_THREAD = Thread(target=accept_incoming_connections,  
args=(server_socket,))
```

- **What goes wrong:** If you call accept_incoming_connections(server_socket) inside Thread(...), Python executes the function immediately in the main thread, never creating the thread properly. By the time the thread would start, the function has already returned (or blocked/exited).
- **How to fix:** Make sure your code is:

python

Copy code

```
ACCEPT_THREAD = Thread(  
    target=accept_incoming_connections,
```

```
    args=(server_socket,)  
)  


---


```

2. Using .join() Too Soon or in the Wrong Place

python

Copy code

```
# Problematic pattern  
  
while True:  
  
    client, client_address = server_socket.accept()  
  
    client_thread = Thread(target=handle_client, args=(client, client_address))  
  
    client_thread.start()  
  
    client_thread.join() #<--- Blocks until client_thread finishes!
```

- **What goes wrong:** The server blocks until that single client finishes, preventing it from accepting further clients. If/when the client disconnects or an exception occurs, the function returns and may cause your entire server to exit (depending on your main loop logic).
- **How to fix:** Remove `client_thread.join()`. Let each client's thread run in the background:

python

Copy code

while True:

```
    client, client_address = server_socket.accept()  
  
    client_thread = Thread(target=handle_client, args=(client, client_address))  
  
    client_thread.start()  
  
    # NO .join()
```

3. Main Thread Exits and Kills Child Threads

If your main thread finishes, Python will shut down all non-daemon threads (in many cases). For example:

```
python
```

Copy code

```
def main():

    print("Server is starting...", flush=True)

    server_socket = create_server_socket()

    print("Waiting for connection...", flush=True)
```

```
    ACCEPT_THREAD = Thread(target=accept_incoming_connections,
args=(server_socket,))
```

```
    ACCEPT_THREAD.start()
```

```
    ACCEPT_THREAD.join()      # If accept_incoming_connections returns, main()
ends
```

```
    server_socket.close()
```

- **What goes wrong:** If `accept_incoming_connections` ever returns (e.g., an exception, a break, or the thread completes), `main()` ends, and Python shuts down. This can produce an unexpected exit code (often -1 if forcibly closed).

- **How to fix:**

1. Remove `.join()` if you want to continue accepting clients indefinitely.
2. Ensure `accept_incoming_connections` is an infinite loop that doesn't break, or handle exceptions so it doesn't exit.
3. If you need the main thread to remain alive, you can do something like:

```
python
```

Copy code

```
ACCEPT_THREAD = Thread(
    target=accept_incoming_connections,
    args=(server_socket,),
    daemon=False # ensure it's not a daemon thread
```

```
)  
ACCEPT_THREAD.start()  
  
# Option A: Let the main thread block forever  
ACCEPT_THREAD.join() # Only if you're sure accept_incoming_connections  
never returns  
  
# Option B: Put the main thread into an infinite loop  
while True:  
    pass  
# (or wait for a keyboard interrupt, or something else to keep the process alive)
```

4. Exceptions in the Thread Not Shown in Console

If an exception occurs inside `accept_incoming_connections` or `handle_client`, you may never see it unless you catch and print it. Consider wrapping the server logic in a big `try/except`:

```
python  
Copy code  
def accept_incoming_connections(server_socket):  
    try:  
        while True:  
            client, client_address = server_socket.accept()  
            print(f"{client_address} has connected.", flush=True)  
            client_thread = Thread(target=handle_client, args=(client, client_address))  
            client_thread.start()  
            # No join() here!  
    except Exception as e:  
        print(f"accept_incoming_connections crashed: {e}", flush=True)  
And in main():
```

```
python
Copy code
if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        print(f"Server encountered an error at main level: {e}", flush=True)
```

This way, you'll see if something is throwing an exception that kills the thread.

5. The Environment Is Buffering or Hiding Output

a. Use `python -u` (unbuffered mode)

On some systems, Python's standard output is buffered, especially when not running in an interactive console. Using `-u` forces unbuffered stdout:

bash

Copy code

```
python -u template_server.py
```

And in your code, you can also ensure each print has `flush=True`.

b. IDE or Terminal Closes When the Script Ends

If you run both server and client in the **same** IDE run configuration, the IDE might automatically stop the server's console once the client script finishes.

Solution: open two separate terminals or run configurations, one for `template_server.py`, the other for `template_client.py`.

c. Check if You're Really Running the Updated Code

Sometimes, especially in large projects or with multiple Python interpreters/virtualenvs, you might be running an old copy. Double-check:

bash

Copy code

```
which python
```

```
python --version
```

```
python template_server.py
```

...and add a distinct debug print in the server code to confirm it's the correct version.

6. Daemon Threads vs Non-Daemon Threads

By default, `Thread(...)` is non-daemon, meaning the main script will wait for child threads to finish before exiting. If you accidentally mark threads as daemon:

```
python
```

Copy code

```
client_thread = Thread(target=handle_client, args=(client, client_address),  
daemon=True)
```

...and the main thread exits, all daemon threads are killed immediately without printing.

Solution: keep threads non-daemon or ensure the main thread stays alive.

7. Testing With a Minimal Example

If you suspect a deeper issue, strip your server down to the absolute minimum:

Server (min_server.py):

```
python
```

Copy code

```
import socket
```

```
from threading import Thread
```

```
HOST = "127.0.0.1"
```

```
PORT = 33002
```

```
BUF = 1024
```

```
def handle_client(conn, addr):
```

```
    print(f"handle_client started for {addr}", flush=True)
```

```
while True:
    data = conn.recv(BUF)
    if not data:
        print(f"{addr} disconnected", flush=True)
        break
    print(f"Received from {addr}: {data.decode()}", flush=True)
    conn.sendall(b"Echo: " + data)
    conn.close()

def accept_incoming(server_sock):
    while True:
        conn, addr = server_sock.accept()
        print(f"Accepted connection from {addr}", flush=True)
        Thread(target=handle_client, args=(conn, addr)).start()

def main():
    print("Minimal Server Start", flush=True)
    s = socket.socket()
    s.bind((HOST, PORT))
    s.listen(5)
    print(f"Server listening on {HOST}:{PORT}", flush=True)
    accept_incoming(s)

if __name__ == "__main__":
    main()

Client (min_client.py):
python
Copy code
```

```
import socket

HOST = "127.0.0.1"
PORT = 33002

s = socket.socket()
s.connect((HOST, PORT))
s.sendall(b"Hello Server")
data = s.recv(1024)
print("Server said:", data.decode())
s.close()
```

- **Server Output** (run in one terminal):

```
java
Copy code
Minimal Server Start
Server listening on 127.0.0.1:33002
Accepted connection from ('127.0.0.1', 56789)
handle_client started for ('127.0.0.1', 56789)
Received from ('127.0.0.1', 56789): Hello Server
```

- **Client Output** (run in another terminal):

```
arduino
Copy code
Server said: Echo: Hello Server
```

If **this** minimal example shows prints but your real server does not, the difference in code structure reveals where the problem lies (thread usage, .join(), environment issues, etc.).

The Bottom Line

When the server immediately exits (exit code -1) or no server prints appear:

1. **Remove join() calls** that block or cause the server to exit prematurely.
2. **Ensure you're passing the function to Thread correctly** (i.e., no parentheses).
3. **Keep the main thread alive** (avoid returning from main() if you need a continuous server).
4. **Run the server and client in separate terminals** so the server isn't killed when the client script ends.
5. **Use flush=True and/or python -u** to guarantee print statements appear promptly.
6. **Wrap threads in try/except** to expose hidden exceptions.

By double-checking these, you'll usually uncover why prints aren't appearing—almost always it's due to the server thread not running as you expect or the process ending sooner than you realize.

Realized that this suggestion- 4. Exceptions in the Thread Not Shown in Console – is probably the cause for the bug.

And did this suggestion -Wrap threads in try/except to expose hidden exceptions.

Next prompt-

gave me the function –

```

221 def my_excepthook(exc): 1 usage  ↳ thelet *
222     #chatgpt code-|
223
224     """
225     Example excepthook function.
226     This is called if a thread raises an unhandled exception.
227     'exc' is a threading.ExceptHookArgs object (Python 3.8+).
228
229     """
230     print(f"Thread {exc.thread.name} encountered an exception: "
231           f"{exc.exc_type.__name__}: {exc.exc_value}", flush=True)

```

" i believe this bug related to an exception that accurse in one of the threads, give instructions on how to debug and ideas for reasons for this bug. prints disappeared right after i modify the MSG header function. check if there's an issue there"

Parts that I used from the response- Thought for 7 seconds

Below is a **fixed main** function that uses Python's **threading.excepthook** properly. In this example, we assume:

1. You have already defined a function `my_excepthook` that logs or handles thread exceptions.
2. You have a function `create_server_socket()` returning a valid server socket.
3. You have a function `accept_incoming_connections(server_socket)` handling new connections.

python

Copy code

```
import threading
```

```
from threading import Thread
```

```
def my_excepthook(exc):
```

```
    """
```

Example excepthook function.

This is called if a thread raises an unhandled exception.

```
'exc' is a threading.ExceptHookArgs object (Python 3.8+).
```

```
    """
```

```
    print(f"Thread {exc.thread.name} encountered an exception: "
```

```
        f"{exc.exc_type.__name__}: {exc.exc_value}", flush=True)
```

```
def main():
```

```
    try:
```

```
        # Assign the global excepthook for threads
```

```
        threading.excepthook = my_excepthook
```

```
        print("Server is starting...", flush=True)
```

```
        server_socket = create_server_socket()
```

```

print("Waiting for connection...", flush=True)

ACCEPT_THREAD = Thread(target=accept_incoming_connections,
                       args=(server_socket,),
                       name="AcceptThread")

ACCEPT_THREAD.start()

# Keep the main thread alive
try:
    while True:
        pass
    except KeyboardInterrupt:
        print("Shutting down server...", flush=True)
    except Exception as e:
        print(f"Error in main loop: {e}", flush=True)

except Exception as e:
    print(f"Exception in main: {e}", flush=True)

if __name__ == "__main__":
    main()

```

Key Changes and Explanations

1. **threading.excepthook = my_excepthook:**

- Must be set **before** starting threads. This ensures any unhandled exceptions in newly spawned threads will be passed to my_excepthook.

2. **Removed ACCEPT_THREAD.join():**

- Using join() on the accept thread blocks the main thread until that thread finishes. If you want a continuously running server, let the accept thread run while the main thread remains alive.

3. Infinite while True with try/except KeyboardInterrupt:

- Keeps the main thread active without blocking, allowing you to gracefully shut down by pressing Ctrl+C.

4. my_excepthook(exc):

- An example function that prints thread exceptions. It receives an instance of threading.ExceptHookArgs (in Python 3.8+). You can modify the logic to log or handle the exception as needed.

This setup will print **all exceptions** raised in threads, including any that occur in accept_incoming_connections or further down in your worker threads.

Another functions that I used chatgpt for –

```
def file_to_json(file_path: str) -> dict: 3 usages ± thelet
    try:
        json_data = {}

        with open(file_path, 'r') as file:
            for line in file:
                # Skip empty lines or lines without a colon
                if not line.strip() or ':' not in line:
                    continue

                # Split the line into key and value
                key, value = line.strip().split(sep=':', maxsplit=1)
                key = key.strip()
                value = value.strip()

                # Convert numeric values to integers
                if value.isdigit():
                    value = int(value)

                json_data[key] = value

    return json_data
except Exception as e:
    raise ValueError(f"Error processing file: {e}")
```

Asked the chat how to get the right type of values from the text instead of the strings, it still didn't work after his modifications.

Rest of the prompts are simple error debugging or basic questions.